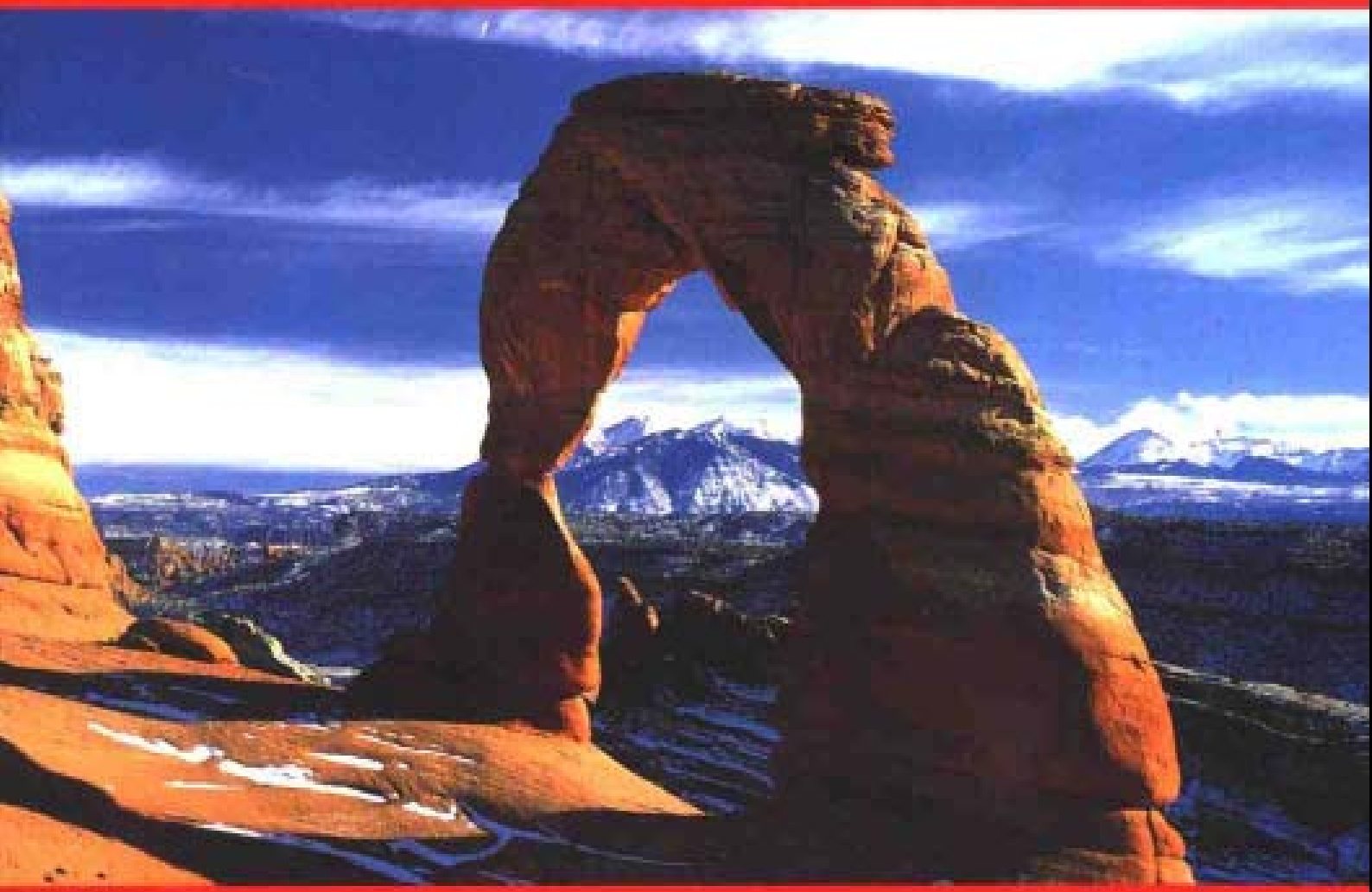
深入C++系列

C++ Strategies and Tactics

C++编程惯用法——高级程序员常用方法和技巧

[美] Robert B. Murray 著
王昕 译



Addison Wesley

中国电力出版社
www.infopower.com.cn

C++ Strategies and Tactics

C++ 编程惯用法——高级程序员常用方法和技巧



在国际象棋中，学会如何移动棋子只能算是入门；要想掌控整个棋局，我们必须了解自己所下的每一步棋后的策略和战术。在C++中也同样如此。掌握正确的策略可以帮助我们避免常见的陷阱，并提高我们的工作效率。在本书中，C++专家Rob Murray就与我们分享了他宝贵的经验和建议，以帮助初中级C++程序员得到进一步的提高。

在本书中，作者大量采用了实际开发中的代码来作为示例，向读者展示了那些有用的编程策略，并对那些有害的做法进行了警示。为了帮助读者更好地理解，在书中的每一章结束前，在该章中被介绍过的主要内容都被放到了一个列表中，此外，书中还给出了一些问题来激励读者们进行更多的思考和讨论。

本书在一开始就向我们讲解了如何为我们的设计选择正确的抽象，提示我们注意抽象和实现之间的区别。然后，我们就将学到如何将已得到的抽象转化成一个（或多个）C++中的类，期间进行的讨论所涵盖的范围上至高层的设计策略，下至底层的接口和实现细节。

接下来本书对单继承和多重继承进行了深入的探索。一开始书中会给出一个关于它们应该用在设计的什么地方的讨论，然后就是一些详细的示例代码，用来向我们演示如何在实践中使用这些概念。对于“如何构建可派生出其他类的类”以及“这么做的好处何在”，书中还专门抽出了一章来讨论它们。

对于C++中新增的模板特性，通过从基础开始到逐步地接触实际应用中的示例，Rob Murray向我们展示了其空前的洞察力。作者同时也向我们展示了多种特定的技巧，以使我们的程序更快、重用性更高，并且更健壮。异常是C++中另外一个新增的特性，对于何时该使用它，何时不该使用它，Murray也向我们给出了他的建议。在本书的最后，我们还可以学到如何将一个项目从C移植到C++之上。书中对该过程的讨论不但包括了其中可能出现的技术问题，也包括了使用技术的“人”的问题。

作者 Robert B. Murray 在撰写该书时是量子数据系统公司（Quantitative Data Systems）中负责软件工程的副总裁，该公司的业务包括向财富500强（Fortune 500）中的公司提供面向对象的软件解决方案。在此之前，他曾经在AT & T的Bell实验室工作，在那里他参与了C++语言、编译器以及库的开发。他同时也是《The C++ Report》杂志的创立编辑（founding editor），主持Obfuscated C++专栏。从1987年起，他就开始在学术会议和专业会议上向人教授C++语言。目前Murray在FNIS公司进行着软件开发的管理，他的兴趣主要包括：程序语言的开发和工具，以及轻量级的开发过程（如极限编程）。

译者 王听 自1995年起接触C++编程，参与过多个C++项目的开发，曾在网上和杂志上发表过多篇有关C++的文章，主要兴趣包括：程序语言的开发、形式化理论、软件开发方法学等。

ISBN 7-5083-1912-5

9 787508 319124 >

责任编辑/陈维宁
封面设计/王红柳

ISBN 7-5083-1912-5
定价：26.00元

深入C++系列

C++ Strategies and Tactics

C++编程惯用法——高级程序员常用方法和技巧

[美] Robert B. Murray 著
王 昕 译



中国电力出版社

C++ Strategies and Tactics (ISBN 0-201-56382-7)
Robert B. Murray
Authorized translation from the English language edition, entitled C++ Strategies and Tactics, published by Addison Wesley, Copyright©1993
All rights reserved.
No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.
CHINESE SIMPLIFIED language edition published by China Electric Power Press Copyright©2003

本书由美国培生集团授权出版。
北京市版权局著作权合同登记号 图字：01-2002-4846 号

图书在版编目（CIP）数据
C++编程惯用法——高级程序员常用方法和技巧 /（美）默里著；王昕译.
—北京：中国电力出版社，2003
ISBN 7-5083-1912-5

Ⅰ.C... Ⅱ.①默...②王... Ⅲ.C 语言—程序设计 Ⅳ.TP312

中国版本图书馆 CIP 数据核字（2003）第 111163 号

丛 书 名：深入C++系列
书　　名：C++编程惯用法——高级程序员常用方法和技巧
编　　著：（美）Robert B. Murray
翻　　译：王昕
责任编辑：陈维宁
出版发行：中国电力出版社
　　地址：北京市三里河路6号　　邮政编码：100044
　　电话：（010）88515918　　传　　真：（010）88518169
印　　刷：汇鑫印务有限公司
开　　本：787×1092　1/16　　印　　张：13.625　　字　　数：297 千字
书　　号：ISBN 7-5083-1912-5
版　　次：2004 年 2 月北京第 1 版　　2004 年 2 月第 1 次印刷
定　　价：26.00 元
版权所有　翻印必究

代译者序

"It is one of those books that every professional C++ programmer should have read as it is full of good advice based on a wealth of experience." ——ACCU 主席 Francis Glassborow

是的，上面的评价就是 ACCU 主席对于你现在手头上的这本书的评价。现在世面上关于 C++的书籍种类之多，可谓浩如繁星，但能够得到 Francis Glassborow 亲自介绍并且给出"**Highly Recommended**"这一最高推荐度的 C++书籍，不过几本而已，而这本就是其中之一。

作为一种支持多种设计范型的通用程序设计语言，C++程序设计语言自身所特有的内涵以及外延都决非一两出色的书籍所能包含亦或一两年艰苦的学习所能掌握的。究其原因就是因为在使用 C++语言的过程中，你会拥有太多的选择。无论是在用 C++进行设计或者实现，做出正确的选择都决非易事。非常多的 C++书籍都只是详细地介绍了语言的特性，即使是某些口碑良好的读本，也同样只是在传教语言本身却忽视了告诉读者如何在使用 C++的过程中去做出正确的选择。而这本书却在这个关键方面有着出色的表现。从这一方面来说，这本书是为那些严肃的以及真正书写 C++代码的程序员准备的。

从本书中，你不会学习到什么是左值，什么是循环，什么是类。因为这本书理想的读者是那些已经具备了一定 C++基础，期望并且正在迈向 C++专家之路的程序员。尽管本书的作者在一开始就已经表明了没有人可以仅仅通过阅读此书而成为专家，但是以我对这本书的观点来看，作者 Rob Murray 无疑是在以自己数十年的专家经验为大家铺设一条通往专家的大道，这也正印证了那句经典——"师傅领进门，修行靠个人"。

那么到底该如何给这本书定位呢？一位使用了 C++数十年的专家写成的一本关于如何在 C++中做出正确选择的专家级读本，这就是我心目中给此书的定位。在购买此书之前，你应该至少阅读过一本关于 C++语言的书籍，但无需对 C++有很深刻的认识，因为这正是此书想要达到的目的。如果你曾经阅读过《C++沉思录》，那么这本书的程度和那一本在某种意义上是一致的，因为它们都不是关于语言介绍的，而都是关于如何使用 C++的。并且同样的在书中遍布了许多关于如何使用 C++进行更好编程的真知灼见。

在这里，我还给大家一点阅读方面的建议。本书本身是按照主题来组织的，也就是说一个章节一般都是独立地说明了 C++方面的某一个主题，并且给出了详细的解释以及说明。这样，阅读本书就并不需要循序而来，你可以一上来就选择你所感兴趣的主题章节进行阅读，不过我还是建议大家在阅读之前首先好好地熟读第一章。这一章是关于抽象（Abstraction）的，关于

抽象这一主题的说明并不十分常见于一般的编程书籍或者语言介绍书籍中。然而，抽象这个与编程有着本质联系的概念对于培养良好的程序观有着不可替代的作用。此书中关于抽象概念的说明虽然不是很深入，却非常直观和易于理解，我想在拥有了一个良好的抽象概念后，再去细细品味书中所提到的其他关于C++的一切是有好处的。

最后，我还想单独地挑出书中的一章来谈谈我的看法。此书第9章标题非常醒目——重用（Reusability），软件工程中的一个核心而又敏感的词汇。在一本C++书籍中，对如此主题进行论述是否合适？这也是我拿到书稿看过目录后的第一反应。因为重用这个话题决不是一两句话、一两个章节可以说清楚的，如此一来，如果作者没有很好的技术功底以及文字能力，这样的章节只可能沦落成“食之无味、弃之可惜”的鸡肋。在我个人阅读过的许多书籍中，这种现象常常见到，以至于现在我会把一本书是否包含了鸡肋章节作为评价一本书好坏的重要标准之一。然而，在详细地读过这一章节之后（这是我在阅读这本书的过程中看的第2章，第1章就是关于抽象的那章。是的，我个人是按照主题来阅读此书的），我想我的疑虑全部都被打消了。作者Rob Murray不愧是程序设计领域内的专家人物，可谓深谙重用之道，而其叙述手段更是精彩，首先从重用的目标以及发现说起，再一步步结合C++语言的实现方式对多个主题进行了深入浅出的论述，其对软工学中重用的概念以及C++的多个实现主题之间的结合可谓完美。在和此书译者王昕先生的私下交谈中，译者多次提到这一章也是让他本人最受益的，一个使用C++已经有八年之久的程序员尚能从中获益，我相信一般读者在深刻理解了本章内容后一定会有同样的感觉。在我眼里这一章的内容就已经对得起读者对此书的所有投资了。

关于此书作者及译者，书底有很好的介绍，我也就不再着墨了。书没有最好的，只有最适合自己的，在我说了这么多之后，这本书是否适合你，还是需要你自己做出判断，但再一次看看Francis Glassborow对此书的另外一条评价也许能让你更快并且正确地做出买与不买的选择，就像这本书将会告诉你更多关于C++使用上的该与不该一样。

“Get a copy, read it and pass it on. Get your own copy if you want to keep ahead of your colleagues.” ——ACCU主席 Francis Glassborow

倪　硕

于 长沙中南大学

序

在专家看来，C++可以帮助软件设计者和程序员构造出模块化、易维护以及高性能的系统。然而，对新手来说，这门语言的复杂程度是一个不小的威胁。在 C++中有着许多不同的语言特性，只有具有一定经验之后，我们才会知道各种特性的适用情况。

本书的目的就是加速读者的这种学习进程。大部分成功的 C++程序员并不会简单地从语言规则中复述出某些章节或是条款，相反，他们总是习惯于自己以往工作中所获得的行之有效的那些惯用法和技巧。我们的目的就是帮助那些 C++的新手学习到那些在实践中最有效的惯用法。同时，我们也会在书中指出那些我们经常接触到的 C++缺陷。

在本书中，我们的讨论并不试图覆盖整个语言，那些有关语言语义的精确定义可以查看参考手册。我们主要帮助读者构造出可以被那些不是 C++语言专家的人所理解的程序。我们不但讨论了那些使程序变得优雅及高效的技巧，也展示了如何使它们更容易被理解和维护的方法。

致谢

本书中的想法和编程惯用法几乎都不是我发明的。我的目的就是把它们（在我 8 年的 C++使用过程中从其他人那学到的并被我认为是最重要的策略和战术）用一种 C++新手也能学会的方式展现给读者。这些收获中的部分来自于实际的开发项目中（这些项目被我们从 C 移植到 C++上）的经验，另外一些则来自于和其他高手的讨论。

关于模板以及函数库设计的许多好的想法（包括那些隐藏在本书所给出的容器类后的想法）都来自于最初由 Martin Carroll、Andrew Koenig 以及 Jonathan Shopiro 所设计的 USL 标准组件库[1]中的类。对于在本书中出现的任何错误，则都由我负责。Andrew Koenig 是一个 C++语言专家，他对我的帮助非常大。另外，那些参与过我所举行过的几期“C++策略与战术”研讨会的人也帮助启发了我书写本书，并对书中的想法进行提炼。其他的重要的想法来自于 Tom Cargill、John Carolan、Jim Coplien、Mark Linton、Gerald Schwarz，当然，还有 Bjarne Stroustrup，要不是他发明了 C++语言，本书也就不可能出现了。

Brian Kernighan 阅读过本书的多次草稿，他极好的反馈意见起了很大的帮助作用。同时我

1 译注：关于 USL 标准组件的文章参见于 1993 年 6 月的《C++ Report》中 Martin Carroll 所写的《Design of the USL Standard Components》。

还要感谢 David Annatone、Steve Buroff、Tom Cargill、Bill Hopkins、Cay Horstman、Lorraine Juhl、Peter Juhl、Stan Lippman、Dennis Mancl、Scott Meyers、Barbara Moo、Lorraine Weisbrot Murray、Bjarne Stroustrup、Clovis Tondo、Steve Vinoski 以及 Christopher Van Wyk，他们对本书早期的草稿提出了意见。此外，Lorraine Weisbrot Murray[2]对我的鼓励、理解、支持以及爱使得这一切努力都成为了可行的。

2 译注： Lorraine Weisbrot Murray 是 Robert B. Murray 的妻子（经 Robert B. Murray 证实）。

目 录

代译者序

序

第 0 章 简介 1

0.1 关于示例代码 1
0.2 本书的组织 2
0.3 问题 2
0.4 语言规则 3

第 1 章 抽象 4

1.1 有关电话号码的抽象模型 6
1.2 抽象模型间的关系 7
1.3 请考虑边界条件 12
1.4 使用 CRC 卡片来辅助设计 13
1.5 小结 13
1.6 问题 14

第 2 章 类 15

2.1 构造函数 15
2.2 赋值 23
2.3 公用数据 24
2.4 隐式类型转换 28
2.5 操作符重载：成员或非成员？ 33
2.6 重载、缺省值以及省略符 36
2.7 Const 37
2.8 返回值为引用 43
2.9 静态对象的构造 44
2.10 小结 45
2.11 问题 46

第 3 章 句柄 48

3.1 一个 String 类 48
3.2 使用计数器来避免多份拷贝 50

3.3 避免进行重编译：Cheshire Cat 55
3.4 使用句柄来隐藏设计 56
3.5 多种实现 57
3.6 作为对象的句柄 61
3.7 综述 61
3.8 小结 61
3.9 问题 61

第 4 章 继承 63

4.1 is-a 关系 63
4.2 公有继承 68
4.3 私有继承 68
4.4 保护型继承 70
4.5 与基类抽象的一致性 71
4.6 纯虚函数 73
4.7 有关继承的细节和陷阱 75
4.8 小结 78
4.9 问题 79

第 5 章 多重继承 80

5.1 作为交集的多重继承 80
5.2 虚基类 85
5.3 一些有关多重继承的细节问题 89
5.4 小结 91
5.5 问题 91

第 6 章 考虑继承的设计 93

6.1 被保护的接口 93
6.2 我们的设计是否应该考虑到继承？ 96
6.3 一些为继承所做的设计的例子 100
6.4 结论 105
6.5 小结 105
6.6 问题 106

第 7 章 模板 107

7.1 模板类 Pair 109
7.2 一些有关模板的细节 110
7.3 模板的实例化 111

7.4 智能指针 113
7.5 作为模板参数的表达式 119
7.6 模板函数 121
7.7 小结 124
7.8 问题 124

第 8 章 模板的高级用法 126

8.1 使用了模板的容器类 126
8.2 示例：Block 127
8.3 有关 Block 的设计细节 130
8.4 带有迭代器的容器：List 134
8.5 迭代器的设计细节 141
8.6 性能上的考虑 144
8.7 对模板参数的限制 146
8.8 模板特化 148
8.9 小结 154
8.10 问题 154

第 9 章 重用 156

9.1 发现和获得 157
9.2 健壮性 158
9.3 内存管理 163
9.4 可选的内存分配方案 165
9.5 传递参数给 operator new 168
9.6 管理外部资源 170
9.7 寻找有关内存的 bug 171
9.8 名字冲突 175
9.9 性能 178
9.10 不要去猜想，而应该度量！ 179
9.11 算法 179
9.12 动态内存分配中的瓶颈 180
9.13 内嵌 185
9.14 Tiemann 法则 187
9.15 小结 187
9.16 问题 188

第 10 章 异常 190

10.1 一个负面的声明 190

10.2 为什么需要异常？ 192
10.3 一个异常的例子 192
10.4 异常只应该用来表述异常情况 194
10.5 理解异常 195
10.6 责任评估 196
10.7 设计异常对象 197
10.8 小结 200
10.9 问题 200

第 11 章 向 C++移植 202

11.1 选择使用 C++ 202
11.2 采用 C++ 203
11.3 设计和实现 204
11.4 开发一个坚实的基础 206
11.5 相关的思考 207
11.6 小结 207
11.7 问题 208

第0章

简　介

C++是一门规模庞大的编程语言。只了解C++规则的程序员就和只知道棋子如何移动的棋手一样（失败）。为了获得成功，还要学习一些相关的法则和策略。

本书所面对的读者是使用C++语言的初级及中级程序员们，他们一方面期待学到更多有关使用这门语言的知识，一方面又没有时间去参加一个大学学期那么长的一门课程。我们假设读者已经了解了一些C++的基础知识——如：什么是成员函数？如何使用public、private以及protected等等。本书的读者没有必要是一个C++专家，在每次提及一些该语言中比较高级的话题（如多继承）时，我们都会给出一个“回顾”栏目对此进行一个简短的总结。我们关注的是那些被证明在实际中十分有效的策略。在本书中，我们将避免多谈理论，而尽量为读者给出例子及实用的建议。

我们尽量保证不偏离惯例。在本书中，没有一个想法或技术会超出语言的基础，它们大多来自于有着多年C++经验的程序员日常的实践中。很显然，最新的想法总是具有最大的风险。我们希望能够避免将C++的初级用户引导到那些想法上去。除了第10章之外[1]，本书中所有的技术都经过了实践的证明。

我们也无意将读者变成C++专家。C++中也包含了一些阴暗的角落，我们会在书中将它们一一指出并教导读者如何避免它们。写出依赖于语言中定义含糊、意义微妙的规则的程序是不好的，即便作者本人清楚它的意义并保证它可以正确运行，下一个来维护这段代码的人也可能未必能够做到这点。比较好的做法是：坚持使用语言中那些被广泛使用及理解的部分来书写程序。

0.1　关于示例代码

本书中的示例代码都通过了编译（或是取自一个通过编译的文件）。为了使代码尽可能短小，我们在此略掉了那些和主题无关的周边代码（如#include指令）。

本书中的示例大多只处理接口，而不是实现。因此，为了清晰起见，大部分的示例代码都略去了类的私有部分：

1 译注：在本书写作过程中，异常还只是C++中的一个新增特性，但到了现在，在C++中异常已经发展得比较成熟了。

```
class Something {
    // 私有部分被省略……
public:
    Something();
};
```

0.2 本书的组织

在第1章中，我们讨论了如何选择一种合适的高阶抽象方式来实现设计，并着重阐述了抽象和其实现之间的区别。

后续的两章主要关注如何在C++的类中实现较高层次的抽象。第2章的讨论涵盖了自高阶的设计策略到底层的接口和实现细节。在第3章中，我们阐述了几种句柄在类实现中的用法。

接下来的三章介绍了继承。第4章中讨论了公用、保护、私用继承的区别，并给出了如何选择使用各种继承方式的参考意见。第5章中的讨论主要针对超过基类不止一个的继承，我们讨论了应该如何使用它以及有关多继承的一些难以掌握的细节之处。在第6章中，我们讲述了一些有关如何构建类的方法，以使得其他人可以把这些类作为他们的基类来发展他们自己的继承体系。

模板，作为C++语言的一个新特性，其功能非常强大，它必将对人们以后编写C++程序造成极大的影响。我们在第7章和第8章中详细地讲解了模板。由于大部分C++用户对于模板还不甚了解，我们开始只是以基础为始，先描述一些用来构建有效模板的技巧，接着才构建各种各样的模板。包括：使用模板来实现智能指针，一些简单的容器，最后是一个接近实际应用的容器（一个List模板）。然后我们才讨论一些关于模板的高级技巧，如：如何对那个List模板进行重新组织，使其更快、更小、并且具有更多功能。

第9章中讨论了如何把一段可工作代码加入到重用库中；重用的困难程度要远远超过其字面意思。我们在本章中也展示了如何使用前几章中介绍的技巧来提高代码的健壮性、使其更容易使用以及运行得更快。

C++中新的语言特性——异常——可以使我们在编写程序时不再需要经常去进行出错情况的检测，但它也很容易被人们给误用。在第10章中，我们讲述了如何使用异常以及不适合使用它的情况。工业界目前也正在学习如何使用异常，这也使这一章显得稍微有点投机。

将一个项目从C平台移植到C++平台所牵涉到的不止是更改一个编译器那么简单：我们必须在项目开发中习惯一种新的开发软件的方式。我们将会在第11章中简单讲述一些有关采用C++开发的项目中经常碰到的技术及人们自身上的问题。

0.3 问题

除本章外，本书中的其他章节都以一个“小结”加一个“问题”作为结束。在“小结”中，

我们将会以列表的方式给出每一章中的主要概念。那些问题是用来促使读者去思考以及讨论有关技术和人们自身方面的问题，以使得他们可以更容易地理解和记忆每一章中给出的内容。问题中的一部分具有“标准”答案，但更多的则没有，我们衷心地希望读者就此进行的讨论会对他们产生帮助。

0.4 语言规则

在本书书写过程中，C++语言的当前“官方”标准文档是 Ellis 和 Stroustrup 所编写的《Annotated C++ Reference Manual》（也经常被简称为 ARM）[2]。ARM 是 ISO/ANSI C++标准委员会（一个由来自工业界以及学术界的自愿者所组成的团体，他们正在制订一个就会被 ANSI 和 ISO 同时采纳的 C++语言标准）用以制订标准文档的基础。除非 ISO/ANSI 委员会明确提出了某些不同之处，否则在本书中所提到的“语言规则”都是基于 ARM 之上的。实际上，（ISO/ANSI 委员会与 ARM 之间的）大部分的区别都微不足道。我们也鼓励所有的 C++用户都能够遵循标准化结果。

2 译注：目前 C++的标准文档是 1998 年 7 月所发布的 ISO 14882 文档（也称 C++ 98）。

第1章

抽　象

数据抽象（data abstraction）是面向对象设计的一个重要概念。数据抽象要优先于面向对象的设计；然而，随着 C++这样直接支持数据抽象的语言变得流行起来，它的应用范围也变得越来越广泛。

抽象数据类型（abstract data type，也称为 ADT）是一种由用户定义、拥有明显不同的两部分的类型：

一个公用的接口（public interface），用于指定用户使用该类型的方式，而一个私用的实现（private implementation），在类型内部被使用，以提供公用接口所给定的功能。

在 C++中，private 和 public 关键字被用于指定类声明中哪部分是实现，哪部分又是接口。通过这种方法，编译器就可以确保类的使用者不会绕过类的接口而直接访问其私用成员。然而，另一个事实就是，类有私用成员也不表示它的设计就很好。

对于 private 和 public 来说，它们本身还存在着一个远比低层次语言规则更重要的思想。这个思想还从未被 C++编译器显式地检测过，但它对于设计模块化、易维护的程序来说却是至关重要。它表述的是一种概念，那就是——每个编程问题的解决方案都可以被划分为两部分：一个抽象模型（abstract model），它是在程序员和程序用户之间都能取得一致意见的、用于描述该问题以及其解决方案的智力模型；对于该模型的实现（implementation），就是程序员使用特定的方法使计算机可以表述出这种抽象。在本章中，我们将会通过对一个类的设计、细化以及记录过程，来逐步体现出抽象的过程。如我们将看到的那样，“简单地书写一页手册，然后让代码来适合它”这种方式在程序设计中是行不通的。

下面是一些理由，用于说明为什么一个经过详细考虑并记录下来的抽象模型对于程序设计来说很重要：

它有助于帮助其他人来理解如何使用你所设计的类。如果你正试图使用一个链表类，你的主要的（也是最初的）考虑应该不是包含这个类的头文件的名字，也不是该类中成员函数的名字以及其参数类型等。相对来说，你会更注重那些有关该类抽象模型的基础问题，如：它提供了什么操作？能不能在链表上进行回退？能不能在常数时间内访问到链表的头和尾？同一对象能不能在一个链表中出现多次？链表中包含的是对象本身呢，还是对对象的引用？如果链表包含的是对象的引用，那么由谁来负责创建及销毁这些对象？当链表中的某个对象被销毁

时，链表应该有什么样的调整？等等。

对这些问题的回答将会直接影响到那些使用该链表类的应用程序的设计。（例如，如果你的应用程序需要能够在链表上双向移动，那么一个单向链表类在此处就不再适合了。）

如果你不能理解并记录下你的类所支持的抽象模型，那么用户就可能会选择不使用你所提供的类（更糟的是，用户决定使用你的类，然后却发现他们做了一个错误的选择）。

抽象模型是你与你的用户之间的一个协议。由于抽象模型对用户的程序设计有着很重要的影响，你将发现对它进行不向上兼容的改动会有多困难，有时那样的改动甚至就是不可能做到的。例如，一个去除掉链表中后退功能的决议，对公开使用的 List 类来说将会是一场灾难。为什么这样说呢？因为某些 List 的用户可能会将其应用程序的设计完全围绕着它能够支持后退这种抽象来进行。这种在抽象模型中的不兼容改动肯定会使某些已有的用户陷入困境。如果一个类已经被广泛使用，那么它的每一个特性都将至少被一个用户所使用，如果你轻率地决定去除掉某个特性，必将在特定的用户群中引起骚乱。

这就意味着，在你决定将你的库发布给客户时，对库的抽象模型进行细化和完善尤为重要。在实现中产生的错误通常都可以很容易地在下一个版本中得到更正；但在抽象模型中产生的问题（除去冗余错误之外）则将持续存在于类的整个生命周期中。

在记录抽象模型的过程中，我们常常可以发现设计中的重要缺陷。在软件项目设计的早期，我们通常会过高地估计我们对于问题以及我们所提出的解决方案的理解程度。在将早期的模糊想法用精确的言语表示清楚的过程（即抽象模型的记录过程）中，我们就能更关注于那些我们以前从未考虑到的方面。毕竟，在认识不清晰的时候，我们是无法提出合理的解决方案的。

清晰的抽象模型文档有助于其他人重新构造出你的类的新版本（这包括继承或是重新实现两种方式）。要想实现出一个能够与现有代码共同工作的新版本的类，你所需做的不止是在成员函数的名字以及类型签名上做到和最初的代码完全匹配；更重要的是，新版本的类必须还能够符合旧版本的抽象模型的要求。

一旦理解了抽象模型，我们就可以避免“以实现来驱动设计”的情况出现。不管其承认与否，许多的软件设计者在设计一个新类的接口时，他们的脑海中都已经有了一个“明显”的实现方案。这就会使得他们在设计中不自觉地将抽象模型向这种实现方式靠拢。这种做法不但不能提供一个用户易于理解的、并且认为就应该如此的接口，它还会使实现的细节遍布于接口之中，导致以后对实现进行改动变得异常困难。

当然，其他极端的做法也同样会导致麻烦的出现（如：设计出一个完全不顾及实现方案的可能性的接口）。一个接口，不管它有多么优雅，如果它不可能被实现出来，或是实现它需要一些让人无法接受的性能上的损失，那么对用户来说它就起不到任何帮助作用。在这些考量之间取得一个合理的平衡是类设计中最具有挑战性的部分。最后说明一点，设计抽象模型和设计实现方案应该是两个独立的行为。但尽管如此，这并不意味着我们需要用不同的人手来分开处理这两部分，重要的是，开发人员必须知道自己在某个特定的时间时，自己到底是在负责抽象呢，还是负责实现？

仅仅考虑抽象（而不是实现），我们有几种常用的方法。在抽象的过程中决定“什么应该有，什么不应该有”是面向对象设计者的一个关键技巧。在下一小节中，我们将开始构造出一个简单的抽象模型。

1.1 有关电话号码的抽象模型

在本节中，我们将对用于表示电话号码的类的抽象模型进行最初的探讨。我们期望用这个类来代指各种电话应用程序（如交换机系统、账单系统等等）中的电话号码。

我们应该怎么开始我们的第一步呢？一个不错的方法就是：用一句话来描述该对象是用来干什么的（注意，不是“是什么”）。这种描述方式应该尽可能的抽象，尽量不要涉及到对象的内部结构。这种“官面总结（executive summary）”应该尽可能的简单，简单到即便是行政人员也可以轻而易举地理解它。

例如，下面就是对于电话号码的一个糟糕的官面总结：

“电话号码包括3位地区号码，后面紧接着3位交换机号码，最后是4位数字。”

这样的描述并没有提及电话号码是用来做什么，或者它和谁交互的问题。相反的是，它简单地把电话号码的结构描述了一遍。这不但没有解释清楚电话号码是做什么的，它还限制了电话号码的范围。例如，通过modem所拨打的电话号码可以含有非数字的字符（如&就用于告诉modem来等待第二次的拨号音，另外还有#和*这两个特殊的按键）。更严重的就是，它还限制了在美国和加拿大之间能够合法通话的电话号码集。

下面的修改就要稍微好一些：

“一个电话号码指定了世界上的某一部特定电话。”

此处我们并没有丝毫提及电话号码的结构，它只是阐述了另一个用于电话号码的抽象。

当你写下这样的句子后，请花一些时间来仔细琢磨它。请像律师对待一份合法的合同那样仔细检查其中的每个单词。这句话中有什么明显的漏洞吗？它所隐含表达的意思又是什么？

让我们先来注意“指定”这个词。它意味着什么呢？一个电话号码是不是惟一指定一部电话呢？换句话说，能不能有两部以上的电话共用同一个号码呢？事实上，在许多的商业机构中，大部分电话都是共用一个电话号码的：在有电话进来时，会有一个本地的交换机将它转到一部空闲的电话上面去。同样，对于一个特定的电话来说，电话号码也不是惟一的。当你拨打555-1234时，对方的电话就取决于你拨打时所处的区域。

这样，我们就得知了：一个电话号码并不代表惟一一部电话，它的含义取决于使用它的区域。为了反映出这种更新，我们进行了如下的官面总结：

“当与呼叫电话关连起来时，电话号码就是决定被呼电话的关键。”

这样，我们就把电话号码与呼出电话的关系给涵盖进去了，并且也隐式地表达了被呼电话

的惟一性。但让我们仔细看看这句话的后半部分。是不是说你拨打电话后就可以得到被呼电话呢？也许如此吧，但更精确的说法是：拨打电话会导致一个到被呼电话的连接的建立。这两种说法之间有什么不同呢？它们之间的不同完全取决于电话用户是否会对连接本身有兴趣。那么，在两台电话间的连接有什么属性会让人们感兴趣呢？当然有了：一个连接有一个开始时间（start time），一个结束时间（end time），以及一定的开销。它们不属于这两台电话，而仅仅是电话连接本身的属性。

拨号必定产生一个被呼电话的结论并不总是正确的：有时线路可能比较繁忙；有时也可能因为某些其他原因会导致呼叫失败……所有这些都不会产生我们所期望的被呼电话。有鉴于此，我们将上面的抽象结论继续细化为：

“当从呼叫电话拨号后，电话号码就是决定可能连接到的被呼电话的关键。”

上面的句子与我们开始时所给出的抽象表述相比，它更贴近实际应用中的抽象模型。然而，要想真正地了解它的含义，我们还必须定义一些它所依赖的其他抽象模型。例如：

“（电话间的）连接表示的是在两部（或多部）电话间的逻辑连接。”

1.1.1 什么不应该出现在抽象模型中？

没有出现在抽象模型中的东西和存在于模型中的一样重要。在我们上面给出的那个电话号码的抽象模型中，我们并没有假设出：一个电话号码有多少位；它是否包含非数字字符；电话与电话号码之间是否是一一对应的以及它们间的连接是如何建立的。

在保持实用性的前提下，我们通过尽可能地将该模型最小化以使其尽可能通用化。我们提高了它应付今后未知变化的能力，以避免频繁地修改我们的抽象模型（但实现细节可能有所改变）。例如：蜂窝电话的出现或者是标准电话系统中新增的按键都不致迫使我们修改我们的抽象模型。

1.1.2 如果存在疑问，暂不考虑

在吃不准某个特殊的概念是否应该包含于抽象模型中时，通常的安全做法是不去考虑它，而只是把它包含入我们的抽象模型中去。这是因为，由于缺少造成的错误通常都可以用一种向上兼容的方法进行修复——一般来说我们只需在类中新增一个成员函数就可以了。扩展接口总是比缩减接口要容易得多。从另一方面来说，从类中去除掉一个成员函数通常很难做到与现有代码兼容，因为可能已经有的用户代码中就调用了这个被去除的函数。

1.2 抽象模型间的关系

作为初始设计过程的部分，设计人员应该仔细地考虑抽象模型与其他模型间的交互，问自己一系列相关的问题。这并不意味着我们需要一个正式的问题清单，根据应用程序的不同，可

能还存在着其他的一些同样重要的问题需要设计人员去早早检测。在本小节中，我们只给出那些经常出现的问题。

问自己这些问题还有着另外一个重要的好处。设计者经常犯的错误就是：这些问题中的部分通常都具有一个看起来十分明显的答案，不过事实上这个答案却是错误的。当你在设计时问自己这些问题时，最好在那些能够很容易就得到答案的问题上停下来，并更加深入地对它们进行思索。这个（显而易见的）答案是不是对的呢？试图去设想一个可以证明它是错误的场景。只要你能够在这种思索过程中找到哪怕只是一个错误场景，你都有可能会因此避免一个将来会给你带来高昂代价的设计错误。

1.2.1 一对一，一对多，多对一，还是多对多？

现在我们来考虑电话和连接之间的关系。一个电话是否可以有多个连接呢？通常的答案都是肯定的，通话中的一方可以接通新的电话而不挂断已有连接。我们同样也要考虑相反的另外一个问题：是否可以有多于一部的电话参与到一个连接中呢？在此处的答案毫无疑问也是肯定的，因为这就是电话的核心用途！当我们关注于这个话题时，我们应该考虑的是，是否可以有多于两部的电话同时参与到一个连接中呢？虽然可能有点会令人感到惊讶，但这个问题的答案同样也是肯定的，电话会议在同一时间时的参与者可以多于三个。因此我们得到的结论是：电话和连接之间的关系是一种多对多的关系。

作为另外一个例子，我们来考虑电话号码与账单地址之间的关系。一个电话号码只能属于一个客户，由此它也只有一个相关的账单地址。然而，同样一个地址却可以有多个电话号码与之相关联。这就是一种多对一的关系（多个电话号码对应于一个账单地址）。

1.2.2 多少才算“多”？

如果连接是多对一或者多对多，那么让我们来想想：究竟这个“多”暗示着多少个对象呢？此时，我们并不需要一个精确的数字，我们想知道的只是，“多”究竟是代表着以下哪一种：

- 二（或者大于它的一个常数数字）；
- 某个常量范围；例如，如果我们讨论的抽象是“一周之内的天数”，那么在一个这样的抽象中，我们最多只能有7个这样的对象；
- 某个变化的范围，大概有好几（百？千？万？）。

需要小心的是：即便是定义得很好的抽象模型，它最终还是会被别人以一种设计者从来都没有考虑过的方式使用。如果我们假设“多数”永远不会超过几十，并由此在实现中使用了一个二次方程式的算法，那么我们的用户在上千个对象的场景下使用我们的实现时，就将得到令人惊讶兼恶心的结果。

1.2.3 如何来保证关系会随着时间的改变而改变？

我们是否必须在对象被创建时确立下来关系的存在呢？当关系被确立后，新创建的对象是

否可以参与到其中呢？

显然，电话号码可以独立于连接而存在；但连接无论在何时都必须至少涉及到一个电话号码。对于连接的另一端的电话，我们对于它有什么要求？是否应该在连接建立的同时就提供它，还是在连接建立后再把它加入连接？从另一方面来考虑的是，一个连接是否可以只涉及一个电话存在？检索（而不是呼叫）电话号码的行为是否会建立一个连接？

回答这些问题需要我们对使用这些对象的应用有着相当的了解。我们需要知道足够多的电话学知识，以此来决定应用程序是否需要在连接正在建立或者中断时对它进行访问。为了讨论方便，我们假定这样的连接有着实际上的作用；例如：计费软件可能需要访问它来记录每次呼叫结束后应该花费的费用。这意味着被呼叫方应该在连接建立时就加入到连接中去。

这个模型同样也可以很好地应用到三方通话中去。此时的连接并不是在同时被建立的，而是随着时间的过去才被建立起来。每次当一个新的成员被加入到通话中时，他们只需要加入已有的连接就可以了。

此时，我们已经对前面的“官面总结（executive summary）”进行了一个细化。那个总结暗示着：一个连接只能存在于两部（或多部）的电话之间。现在我们认识到，那种说法并不全对：我们对于那些可以通过电话网络在一组电话（一部或多部）间相互交谈的方式更感兴趣。于是，我们又可以将总结改为如下形式：

“连接表示的是一组（一部或多部）电话间通过电话网络相互进行交流。”

基于我们对它的不断深入了解，得到这样的抽象细化是一件很正常的事情。在此例中，“连接”这个名词仍然有其意义；但它并不总是对的，我们应该随时准备，在原有的名字不能充分地阐述其意义时为抽象另取一个合适的名字。

1.2.4 Is-A，Has-A，还是 Use-A?

不同的抽象以不同的方式相关联。我们可以把三种最常见的关系归纳为：is-a、has-a 以及 use-a：

is-a关系

当两个类之间存在着下述关系时，我们就说它们之间存在着 is-a 关系：其中一个类所描述的对象属于另外一个类所描述的对象集。例如：Studebaker 是一种（is-a）Car。

is-a 关系的另外一种说法就是子类型化。子类型（subtype）是某些更为通用的类型［也称为超类型（supertype）］的特殊化；子类型的对象同样也是超类型的对象。这些术语源自于 Smalltalk。许多 C++程序员则使用 C++中具有相同意义的两个词：派生类（derived class）来代替子类型，基类（base class）来代替父类型。

例如：“按键电话”和“电话”之间的关系是什么？每部按键电话同样也是一部电话。按键电话只是电话中的一个子集（图 1.1）。

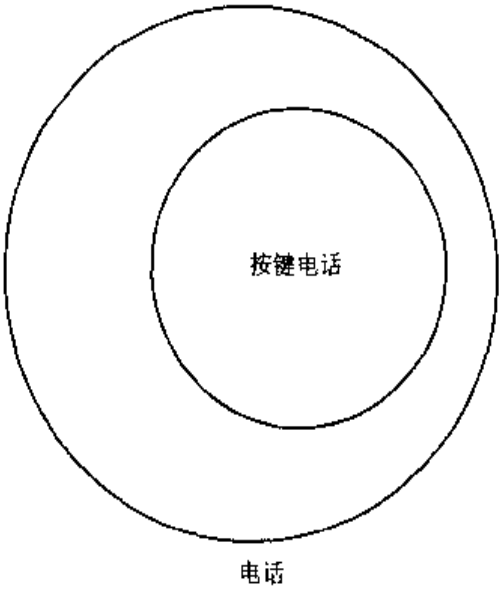


图 1.1 按键电话 is-a 电话

在 C++中，我们使用公用继承来表示 is-a 关系；类 Push_button_phone 应该派生自类 Telephone：

```
class Telephone {
    // Telephone stuff...
};

class Push_button_phone : public Telephone {
    // Push_button_phone stuff...
};
```

子类型（或者说派生类）可以对父类型（或者说基类）进行扩展；一部按键电话可以做一些其他电话做不到的事情。但子类型永远也不能对父类型有所限制（例如：在派生类将一个在基类中的公用成员函数改为私用）；同样，电话能够做到的事情，按键电话都应该能做到。（如果不是这样的话，按键电话也就不应该是电话中的一个子集。）

has-a关系

has-a 关系意味着包含，如果在概念上事物 A 包含事物 B，那么 A 也就拥有 B。电话包括一个扬声器和一个麦克风；按键电话还包括一个键盘。

与 is-a 不同的是，在 has-a 关系中，没有一个对象是另外一个对象的特例，取而代之的是，一个对象是另外一个对象的一部分。一个 Push_button_phone 并不是一个 Keypad，而是拥有一个 Keypad。

C++中的 has-a 关系的通常实现方式是将被包含的对象作为包含它的对象的一个成员。然而，实际上我们并不一定需要这样做。虽然在概念上来讲，Keypad 被包含于 Push_button_phone 中，但这并不意味着我们在实现这种关系时必须使用真正的包含。在实际中，总有很多好的理由让我们不必那么做（我们将在第 3 章讲述这一点）；数据抽象的一个好处在于，对象的实现

方式并不需要完全与抽象结构相匹配。我们不应该把这种方式（指用成员来实现 has-a 关系）来作为主要的考量标准，因为我们所处理的是抽象之间的关系，而不是它们的实现细节。

use-a关系

use-a 关系应用得最普遍：在这种关系中，没有一个对象会是另外一种对象（is-a），也没有一个对象包含另外一个对象（has-a）。取而代之的是，这两个对象仅在程序中某些点进行简单的联系。在我们的电话例子中，电话和连接间具有的就是 use-a 关系。

相互使用的对象间通常通过调用彼此的成员函数来进行联系。当然，还存在着其他的联系方式（如：共享内存，或者是某种消息传输机制）可以做到这一点。

1.2.5 关系是单向的还是双向的?

给定两个对象之间的一个关系，我们是否就可以通过其中的一个对象来得到另外一个对象的信息呢？也就是说，如果我们已经知道了电话和连接之间的关系，那么我们是否可以从连接中得到有关电话的信息，或者是从电话中得到连接的信息呢？这个问题的答案对于编写实现该抽象模型的程序的性能来说，起着本质的影响作用。

单向关系可以用 C++中的指针来实现。给定一个对象，你就可以通过该指针得到另外一个对象。这种行为的反转（从一个对象得到指向它的对象）则非常困难；因为它通常都涉及到从一大堆对象中进行检索，判断一个对象是否正好指向给定的那个对象。根据具体情况的变化，这种做法有时也并不完全那么不好，导致的开销也不那么大，或者是完成的困难程度并不是不可能达到的。

双向关系的使用要简单得多了：给定任意一个对象，我们都可以直接地找到另外的那个对象。然而，建立一个双向关系的开销也要比建立单向关系的开销大得多。我们需要对这两个对象同时进行一定程度的修改，双向关系可能会导致要求更多的空间，更多的运行时间；这一点在关系随着时间同时变化时更为明显。改变一个单向关系可能只涉及到一个指针的改变。由于往两个对象中增添新的关系可能会导致已有关系的失衡，改变一个双向关系的操作可能涉及到三个（甚至更多的）指针变化。在两个对象间，单向关系要显得更快、更小，也更容易被编程实现，但它却限制了用户使用抽象模型的能力。这也是实现细节可以（也是必要）影响设计思路的一个常见的地方。

再回到我们关于电话的那个例子中来，电话和连接之间的关系在大多数情况下都应该是双向的：只要给定一部电话，我们就可以找到连接端的另外一部电话。如果经验不足，我们则可能会采取另外一种方法，那就是：查找每一部已有的电话，判断它们是不是处在同一个连接中。

通常我们的选择都不是那么简单明了。例如，考虑一个编译器，在其中表达式被作为一个树状的结构来处理，我们可以将 a = b + c 表示成图 1.2 中所示的树。那么在树中，节点和它们的子节点之间的关系是单向的还是双向的呢？在实际中，为了提高效率，这个关系通常都是单向的，程序也可以以此来编写。例如，如果我们都是自顶而下地来处理每棵树，那么在树中保

留一个指向父节点的指针则显得有点多余。

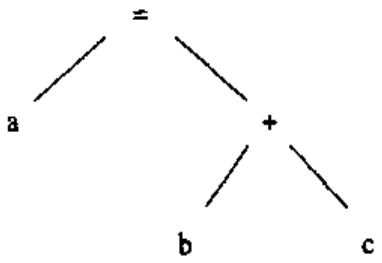


图 1.2 表达式树

1.2.6 两个对象中是否可以有着多重同样的关系?

给定两个对象间的关系，我们是否可以在这两个对象中保留多份该关系呢？例如："父亲"这个关系只能在两个对象间存在一份，一个对象只能是或者不是另外那个对象的父亲。诸如"他是我父亲父亲"这样的表述显得毫无意义。而对于关系"文章引用的书籍"来说，当文章中引用同一本书中的内容超过两次后则可以存在多份这样的关系。

1.2.7 关系是否是必需的?

对于每个对象来说，关系中是否就一定要至少包含一个对象呢？例如，如果有一个用来表示职工的类 Employee，那么关系"向……汇报"是否就一定要包括所有的职工呢？对于部门的主管来说，它应该如何处理？我们的类的结构是否支持主管向整个部门进行汇报的功能？把主管的汇报对象改为空（或者是他自己）会不会更容易一些？

1.3 请考虑边界条件

你可能会对我们为什么需要考虑部门主管这种情况感到惊讶。毕竟，大部分的职工还不是部门主管。现在在设计中考虑这种特殊的情况会不会显得有点早呢？

边界条件存在的地方总是不能满足抽象模型的假设基础。它们可以帮助我们更加关注这些假设条件。如：假设条件是否有效？在何种情况下，假设会失败？是否存在着一种合理的解决办法来处理这些情况？举例来说，如果我们假设每个职工都向另外一个职工汇报工作，那么什么时候这种假设会失效？在这种情况下，它的结果又将如何？

边界条件通常都会导致程序 bug 的产生，而且这些 bug 也不是很容易被及时发现。与其在事发后再来寻找原因，还不如早早地把边界条件列入我们的考虑范围。早早考虑并不意味着我们需要在设计阶段就把每个边界条件的行为都固定下来。随着设计的进行，对问题的理解加深，我们可能会对如何处理部门主管的行为进行修改。但是，为了避免在系统测试或者发布后再来解决这个问题，我们最好还是在前期设计就将边界条件纳入我们的考虑范围。

1.4 使用 CRC 卡片来辅助设计

由 Kent Beck 和 Ward Cunningham 设计的 CRC 卡片是一种用来帮助设计人员在设计阶段前期进行设计的工具，此时设计人员应该做的事情就是：从问题域中找到合适的对象并理解它们之间的关系。

CRC 卡片的含义十分简单，也没有什么技术含量在里面。当一群人聚在一起开始进行高层设计时，他们会为每个潜在的类建立一张索引卡片，在该卡片上面，他们会写上类的名字，它主要的功能，以及它将会和哪些类进行交互。

有了这些卡片后，设计者就可以利用它们来模拟不同的场景。每个参与者负责其中一个或多个对象的角色，然后描述他的对象在面对其他对象的请求时应该做出的反应。通过这种角色模拟，设计者就可以进一步细化系统中类的职能和协作，并把它们记录在卡片上。整个设计小组可以也基于这种讨论来为那些有着不精确(或者是模糊)含义名字的类取一个更贴切的名字。

由于卡片上面的空间有限，这就鼓励设计者们尽可能采取简单的抽象模型——我们很难在一张索引卡片上面描述出复杂的抽象模型。这种卡片也有助于设计者将类进行分类，以构造合适的继承体系。我们可以将那些获得大家一致通过的卡片放到桌子上面，并接着对那些剩下的卡片进行重复的讨论，直到大家取得一致意见为止。

进行这样的角色模拟有时可能会出现一些偏离项目的话题，但它同时也鼓励大家进行“头脑风暴（brainstorm）”，将自己认为想的比较好的东西在非正式的场合中提出来和大家一起讨论。虽然大部分的离题讨论都将无疾而终，但有时偶尔的一两个火花也会产生那些在传统且正式的设计复审时产生不了的突破。

CRC 卡片并不能代替完备的、用来描述类的行为细节的文档。它们主要的作用也就存在于设计的初始阶段：它们可以帮助设计人员正确地得到高层框架的信息，更早（而不是更晚）地理解和分辨出那些重要的问题。

注意：它们不可能像索引卡片那样被整理，你可以将它们粘在墙上，并保留对它们进行重新排列的权利（就像你使用 Post-It 随手贴一样）。

1.5 小结

- 设计抽象模型和设计实现细节应该是两个独立且相关的行为。
- 没有出现在抽象模型中的东西和出现在其中的东西一样重要。
- 如果存在疑问，先不要去考虑它。添加新的功能通常都不会有兼容性的问题，但是去除一个已有的功能则可能会导致代码失效。
- 彻底地检测和记录下设计中的关键点。
- 设计是一个迭代的过程。

- 请考虑边界条件。
- 在设计的初始阶段使用 CRC 卡片。

1.6 问题

1. 我们的电话号码的抽象模型是否还需添加其他新的抽象模型来更好地完善它？

2. 如果拨号由于某种原因（例如线路忙）失败了，那么应该有什么发生呢？我们的抽象模型在这种情况下的反应应该是怎么样的呢？

3. 我们的 Connection 类假设至少有一部电话被加入到连接中去。现在我们考虑对它进行一些修改以适应下面这种情况：在创建 Connection 对象时，我们并没有给出任何的电话对象。在这种情况下，呼叫是如何产生的？这样做是使得我们的抽象模型更简单了呢？还是更复杂了？

4. 从下面的分组中找出具有 is-a 关系的对来：

- 生物学家
- 罪犯
- 人
- 科学家
- 活体解剖论者

5. Child 和 Father 之间的关系是一对一，一对多，还是多对多？对于这个问题的回答，你主要应该阐明的条件和假设是什么？

第 2 章

类

在类的设计中，最重要的一部分就是对该类所表述的抽象模型有着一个清晰的理解：它和谁交互，它能做什么？我们在上一章中就关注于这个话题。一旦我们得到了这样的一个抽象模型，那么下一步我们要做的就是进行详细设计和类的实现了。在本章中，我们将围绕在这个阶段中的一些最常见的惯例以及 bug 进行讲述。

2.1 构造函数

构造函数中有着比我们所看见的还要多的细节。除了程序员所编写的代码之外，构造函数还可以调用其他的构造函数来初始化对象中的基类对象和数据成员。即使程序中并没有明确的调用，编译器也可以向代码中插入构造函数的调用代码（例如：当程序中有着隐式的数据转换时）。在本节中，我们将关注一些和构造函数相关的常见问题，这些问题中的部分会导致程序的执行速度变慢，另外一些则可能会导致程序中 bug 的产生。

在阅读本章时，你最好能够时刻记住“初始化”和“赋值”这两者之间的区别（详情参见下面的“回顾”）。你同时最好也不要被“缺省构造函数”（不需要任何参数就可以被调用的构造函数，它要么是在声明时就没有参数，要么就是在声明时它的所有参数都被赋予了缺省的值）和“缺省的复制构造函数”（由编译器自动合成的复制构造函数）这些术语给迷惑住。

回顾：初始化和赋值

在 C++中，当一个新对象被创建时，会有初始化操作出现；而赋值是用来修改一个已经存在的对象的值的（此时没有任何新对象被创建）。

```
Thing t = x;      // 初始化（有新对象被创建）
t = x;            // 赋值（已有对象的值被改变）
```

初始化出现在构造函数中，而赋值出现在 operator=操作符函数中。

C++中还有着一种特殊的初始化：用同一个类产生的另外一个对象的值来为将要被创建的对象进行初始化。执行这样的初始化操作的构造函数也被称为复制构造函数，它们通常都有着

如下的格式：X::X(const X&)。

```
// 一个复数类：
class Complex {
// 此处忽略细节
public:
    Complex(double, double);
    Complex(const Complex&);    // 复制构造函数
};
```

如果我们没有在类中声明一个复制构造函数，那么编译器就会为我们合成一个。这个缺省的复制构造函数会将新对象中的每个数据成员初始化为源对象中相应成员的值。

当我们使用传值的方式来调用函数时，编译器会自动产生调用复制构造函数的代码。被生成的临时对象在函数返回时将会（通过调用析构函数）被摧毁：

```
double abs(Complex);    // 使用传值（而不是传址）的方式向函数传递参数
Complex c(0.0, 1.0);
double d = abs(c);
```

在上面的代码中，我们通过调用 Complex::Complex(const Complex&)传递了 c 的一个拷贝给 abs。当 abs 返回时，这个拷贝将会被 Complex 的析构函数（如果有的话）所摧毁。

2.1.1 缺省的复制构造函数的行为是否符合我们的要求?

当我们编写了一个新类时，请注意这两点：缺省的复制构造函数和赋值操作的行为是否符合我们的预期要求。如果不是的话，我们就将不得不声明和定义我们所需要的这两个函数。

通常（但不是所有的）情况下，当对象的所有状态都存储在对象中时，缺省构造函数的所作所为都和我们所预期的一样。例如，我们来看看复数类 Complex 的实现，它的对象中用两个 double 存储了复数的状态：

```
class Complex {
private:
    double real;
    double imag;
public:
    Complex(double r, double i) : real(r), imag(i) {}
// 此处忽略细节
};
```

由于我们没有明确声明复制构造函数，因此编译器将会为我们合成一个缺省的复制构造函数。它的行为就是复制这两个数据成员，而这正是我们这个类所期望的操作。

在另外一方面，假设我们有一个类 String，在其中有一个 char*的数据成员，我们用它来指向 String 类对象所代表的字符串：

```
// 在String.h中:
class String {
private:
    char* data;
public:
    String(const char* cp = "");
    ~String() { delete [] data; }
};

// 在String.c中:
String::String(const char* cp)
: data(new char[strlen(cp)+1]) {
    strcpy(data,cp);
}
```

在上面的例子中，String 类的缺省复制构造函数将会仅仅对那个指针进行复制，最终导致两个 String 对象指向同一块内存。这种结果并不是我们期望的，一旦第一个 String 对象被摧毁，那么被指向的内存也将同时被释放，这时剩下来的那个对象将发现，它所拥有的指针成员指向的是已经被释放了的内存！

如果缺省的行为和我们预期的不一样，我们就必须明确地声明和定义一个复制构造函数：

```
// 在String.h中:
class String {
private:
    char* data;
public:
    String(const char* = "");
    String(const String&);
    ~String() { delete [] data; }
};

// 在String.c中:
String::String(const String& s)
: data(new char[strlen(s.data)+1]) {
    strcpy(data,s.data);
}
```

这样我们就可以确保每个 String 都将拥有一份数据的私有拷贝。

在某些情况下，即使有新对象被创建时，由于某些特殊的操作的缘故，我们也不能使用缺省的复制构造函数。假设我们有一个类 File，我们用它来表示一个文件描述符（也就是一个用于文件 I/O 操作的句柄）。有些应用程序可能需要知道在某个给定的时间内到底存在着多少个 File 的对象（这样做可能是为了避免打开的文件数超过允许的文件描述符上限）。我们可以很容易做到这一点：只要让 File 维护一个当前存在的 File 对象个数的计数器就可以了。在 File 的构造函数中我们会对这个计数器进行递增，在析构函数中对计数器则是递减：

```
// 在File.h中:
class File {
    static int open_files;
// 此处忽略细节
public:
    File(const String& filename, const String& modes);
    File(const File&);
    ~File();
    static int existing() { return open_files; }
};

// 在File.c中:
int File::open_files = 0;

File::File(const String& filename, const String& modes) {
    ++open_files;
    // 此处忽略细节
}
```

静态成员函数 File::existing()将返回现有的 File 对象的计数。

为使这样一个方案生效，每个 File 构造函数都必须更新计数器。缺省复制构造函数不做这个工作，所以我们必须自己来写：

```
File::~File() {
    --open_files;
    // 此处忽略细节
}
```

上面代码中的静态函数 File::existing()将会返回当前存在的 File 对象的个数。

对于缺省复制构造函数是否能够工作这个问题，我们并没有一个通用的规则。一种从经验中得到的方法就是：对那些包含指针的类要“另眼相待”。如果被指向的对象是“属于”该产生的对象，那么缺省的复制构造函数就有可能是错误的，因为它只是简单地复制了指针而不是指针所指向的对象。

2.1.2 复制构造函数不可忽略

即使我们的代码从来不会去调用复制构造函数，我们也不能忽略掉它。请记住，那些使用我们所编写的类的用户可能会去调用类的复制构造函数(他们有可能是通过创建新对象来显式地调用它，也可能是通过用传值方式向函数传递参数来隐式地调用它)。如果确实因为某些原因，使得为类实现复制构造函数变得非常困难，那么请把它声明为私用的，并且不要为它提供任何的定义：

```
class Cant_be_copied {
private:
```

```
    Cant_be_copied(const Cant_be_copied&); // 没有定义体
// 此处忽略细节
};
```

一旦我们这样做了，我们至少可以确信那些无意间调用到复制构造函数的代码将会被编译器（用户代码）或者连接器（类成员或者友元代码）给找出来。虽然这样做并不是很好，但比起那些默默地执行编译器合成的缺省复制构造函数的错误代码来说，它要好得多了。

2.1.3 类成员的初始化

当类中的某个数据成员本身也是一个类对象时，我们应该避免用赋值操作来为该成员进行初始化：

```
class Employee {
private:
    String name;
public:
    Employee(const String&);
};

// 效率不高的构造函数：
Employee::Employee(const String& n) {
    name = n;
}
```

虽然这样做构造函数也能得到正确的结果，但它的效率却不能达到它本来应该达到的标准。当一个新的 Employee 对象被创建时，成员 name 先将会被 String 的缺省构造函数所初始化，然后在 Employee 的构造函数中，它的值又会因为赋值操作而再一次改变。这是两个不同的步骤，不过我们可以把它合并到一个步骤中去：我们可以通过使用初始化语法（initialization syntax）来显式地为 name 进行初始化（详情参见下面的回顾）：

```
Employee::Employee(const String& n)
: name(n) {
}
```

回顾：成员初始化

缺省情况下，在构造函数的函数体被执行前，对象中的所有成员都已经被它们的缺省构造函数所初始化了。那些没有构造函数的成员则将拥有一个未定义的初始值。

编写构造函数的人可以对这种行为进行更改，方法是：在构造函数定义中的参数列表结束的括号后面增添一个冒号以及一个初始体（initializer）列表。每个初始体都包括一个名字以及一个参数列表，这其中的名字就是要创建的类中成员或其基类的名字：

```
class String {
```

```
public:
    String();
    String(const String&);
    // 此处忽略细节
};

class Employee {
private:
    String name;
public:
    Employee(const String&);
};

Employee::Employee(const String& nm)
: name(nm) {
// ...
}
```

下面的代码会告诉编译器只使用一次函数调用来初始化 name 这个成员：

```
String::String(nm);
```

那些内建类型的成员也可以用这种语法来进行初始化，此时的参数列表必须是一个用来指定初始值的简单表达式。

现在这个构造函数只需要进行一次关于 String 的操作（也就是一次初始化），而原来的那个则需要两次（一次初始化以及一次赋值）。在我的计算机上面，对 Employee 的构造函数进行这样的改变可以获得大概 30%的效率提升。

当我们在编写构造函数的定义时，请在写完正式的参数列表后停下来一会，想一想有多少成员可以使用构造函数的初始化语法。通常我们都可以发现所有的成员都可以用这种方法来进行初始化，而当我们写到构造函数的函数体时，将会发现其实我们什么也不需要做！这是一个好的信号：我们的构造函数已经为我们类的成员们选择了一个正确的初始值。

不是类对象的成员

初始化语法同样也可以用来对不是类对象的成员进行初始化：

```
class Employee {
private:
    String name;
    int    salary;
public:
    Employee(const String&, int);
};

Employee::Employee(const String& nm, int sal)
```

```
: name(nm), salary(sal) {
}
```

由于内建类型没有构造函数，使用初始化语法来对整型成员 salary 进行初始化并不能获得比赋值更高的效率，但这样做（使用同样的方法来为所有的数据成员进行初始化）会使得代码的可读性更高。

成员初始化的顺序

C++中规定，一个类中成员的初始化顺序和它们在类中被声明的顺序（而不是构造函数定义中的顺序）必须是一致的。通常情况下，这种顺序问题都不会有什么影响，但在某些场合下，它将导致产生问题——例如：某个成员的初始化过程中使用了另外成员的值。

让我们来考虑一个改进后的 Employee 类，它包括 Employee 的名字和身份认证代码：

```
class Employee {
private:
    String  name;
    long    id;
public:
    Employee(const char* name);
    Employee(long id);
// 此处忽略细节
};
```

我们编写了两个构造函数，一个以职工的名字作为参数，一个以职工的身份认证代码作为参数。每个构造函数都会从一个职工花名册中查找参数中没有给出的那份信息。

下面是我们对构造函数定义的一种（不正确的）尝试：

```
extern String lookup_employee(long);
extern long lookup_employee(const String&);

Employee::Employee(const char* n)
: name(n), id(lookup_employee(name)) {
}

Employee::Employee(long i)
: id(i), name(lookup_employee(id)) { // 运行时错误
}
```

上面的第二个构造函数可以顺利地通过编译，但却不能正常工作。因为在类声明中，成员 name 出现在成员 id 的前面，这确保它每次都会被第一个初始化——即便是我们在构造函数的定义中把 id 的初始化代码写在它的前面！这意味着在第二个构造函数中，lookup_employee 实际上在 id 被初始化前就被调用——这时它的参数也就将是一个毫无意义的随机数值。在此例中，这个问题很容易被解决：我们可以使用外界传递过来的参数，而不是类中的成员：

```
Employee::Employee(long i)
```

```
    : id(i), name(lookup_employee(i)) { // OK
    }
```

但不是每次我们都可以这么容易地解决问题；在某些情况下，我们可能不得不对类的定义进行重新整理。如果成员的顺序意义重大的话，我们最好在头文件中把它记录下来，这样在随后的对类的声明进行重新排列过程中，我们并不会导致某些构造函数的工作出现问题[1]。

编译器忽略掉构造函数中的初始体顺序这个现象看起来好像有违我们的直觉，但这却是C++要求“对象的析构过程必须和其创建过程相反”得到的结果。如果构造函数中指定了一个特殊的构造顺序，那么析构函数将不得不去查询构造函数的定义，以获得如果对成员进行析构的顺序。由于构造函数和析构函数可以在不同的文件中定义，这就将给编译器的实现者造成一个难题。更糟糕的是，一个类可以有两个或者更多的构造函数，对于这些构造函数的定义，我们并不能保证它们中的成员出现的实现都是一致的。然而，我们可以确保类的声明在所有的文件中都将有效，并且不同的文件中出现的声明都是一致的（否则就将造成整个程序的行为无法预期）；所以我们使用类的声明来解决成员的构造和析构顺序。

类成员的引用

如果在类中的某个非静态数据成员是一个引用的话，因为所有的引用都必须被明确地初始化，所以我们必须在该类的每个构造函数中都使用初始化语法：

```
class Payroll_entry {
private:
    Employee& emp;
public:
    Payroll_entry(Emplyee&);
};

Payroll_entry::Payroll_entry(Emplyee& e)
{ // 编译错误："emp"必须被初始化
}
```

此时我们必须问自己，为什么我们要将 emp 声明为 Employee&，而不是 Employee*呢？除了语法上的不同之外，将 emp 声明为引用而不是指针将会在两方面对它的使用造成限制：

- 我们必须在创建 emp 时就为它绑定一个职工（并不存在着一个为“0”或者为“null”的引用）；
- 一旦我们将它和一个职工绑定后，emp 就不可能再被用来绑定到另外一个职工上面。

通过将 emp 声明为引用，如果我们的代码中试图违反上面的两条规则，编译器就将用编译期的错误来帮我们找到它们。使用引用而不是指针从编译器那能够获得的只是编译期的检测，它们实际产生的代码其实并无区别。

1 译注：在这作者显得有点乐观，他假设修改类的开发人员会去认真地阅读头文件中的注释，只有在读过头文件中的注释并正确理解其中意义的前提下，我们才可能得到作者所期望得到的结果。

2.2 赋值

和复制构造函数一样，如果我们没有声明赋值操作符函数，C++编译器会自动为我们合成一个。缺省的赋值操作符会为每个数据成员进行赋值。缺省的赋值操作符的适用场合和缺省的复制构造函数一样。如果缺省的复制构造函数是错误的，那么缺省的赋值操作符几乎也可以被确定为错误的（反之亦然）。

赋值操作符和复制构造函数通常都有着几乎同样的逻辑。它们之间的主要区别在于：

- 当对象被自赋值时，赋值操作符必须可以工作；
- 每次对赋值操作符的调用都会改写一个已经存在了的值。如果在被改写的对象中使用了对象外的资源，那么我们可能需要释放这些资源；
- 赋值操作符可以向外返回一个值。

下面是 String 类的复制构造函数和赋值操作符：

```
String::String(const String& s)
: data(new char[strlen(s.data)+1]) {
    strcpy(data, s.data);
}

const String& String::operator=(const String& s) {
    if (&s != this) {
        delete [] data;
        data = new char[strlen(s.data)+1];
        strcpy(data, s.data);
    }
    return *this;
}
```

请注意上面的赋值操作符。其中对于&s 和 this 的比较是为了防范如下的代码：

```
String s;
s = s;
```

如果没有上面的比较，对于 String 对象的自赋值，我们将会先删除 String 中的数据，然后又试图对这些刚被删除的数据进行拷贝。我们在编写赋值操作符时，必须时刻考虑到自赋值的可能性，并确保自己的代码在这种情况下可能做出正常的反应（通常情况下是什么都不做）。我们可能不期望其他人编写这样的代码：

```
s = s;
```

但不管是使用引用还是指针的代码，我们必须保证它们拥有同样的效果。

一旦我们确信正在处理两个不同的对象，那么与复制构造函数不同的是，赋值操作符必须将原有值中所控制的资源释放掉。（这也就是我们代码中 delete 的作用。）在这之后，除那个返

回值之外，赋值操作符的行为通常都和构造函数中的一致。我们通常都会把那个公有的逻辑提取出来放到一个私用的成员函数中去，并在复制构造函数和赋值操作符中调用这个函数；在本例中，这个公有的逻辑只是一个简单的函数调用，所以我们就直接复制了这个调用。

2.2.1 operator=的返回值

赋值操作符应该返回一个被赋值对象的常量引用。这使得用户可以写出如下的代码：

```
Complex x,y;
// ...
x = y = Complex(0,0);
```

被重载的操作符和内建的操作符有着同样的优先级和结合顺序。=是右结合的，所以下面的赋值：

```
y = Complex(0,0);
```

将会被首先执行，它所产生的结果又将会被赋值给 x。

通过返回一个常量引用，我们就可以阻止将赋值结果作为左值的做法：

```
(a=b)=c;
```

这样做不但看起来比较奇怪，它所产生的结果同样也会让人大吃一惊。例如，假设 a 和 c 都引用到同样的目的，这样做会导致什么结果呢？

让赋值操作返回一个值也有着一个缺陷：即使我们并不使用它，我们也需要用一些代码来返回这个值：

```
a = b; // 返回值未被使用
```

在实践中，这种做法几乎都是可取的。那些额外的代码通常都是一条简单的机器指令（记住：我们返回的是一个引用，它实际上也就是个指针，而不是一个全新的对象）。在大部分情况下，由这一条指令带来的性能影响是可以忽略不计的。如果真的是十分注重效率的话，那么赋值操作符应该被定义为 inline；在此时，任何一个还过得去的优化器都可以优化掉这条没有被使用的指令。

2.3 公用数据

假设有如下一个复数类：

```
class Complex {
public:
    double real;
    double imag;
    Complex(double r, double i) : real(r), imag(i) {}
```

```
    // 其他的被忽略
};
```

这个类可能可以工作，但它的接口却有着重大的缺陷。问题出在那两个公用的数据成员：real 和 imag 上面。这个接口保证 Complex 对象中的实部和虚部会以浮点数的形式存储在对象中。由于用户代码可以直接存取到公用的数据成员：

```
Complex c(3.0, 4.0);
double d = c.real;
c.imag = 0.0;
```

所以一旦我们需要改变信息的存储格式或存储位置时，这样的改动会变得相当困难。

假设在后面的某个版本中，我们希望将 Complex 的实现（而不是接口）中的信息存储方式由笛卡儿坐标格式改为极坐标格式。（也许我们有着的某些算法在处理使用极坐标格式表示的数字时效果最好。）如果笛卡儿坐标是公用数据，我们就将陷入泥潭：我们将不得不改变所有的用户代码或者是在每个 Complex 对象中同时维持极坐标和笛卡儿坐标格式。

如果我们在一开始就避免使用公用数据，我们就不会碰到这样的问题了：

```
class Complex {
private:
    double real_d;
    double imag_d;
public:
    Complex(double r, double i) : real_d(r), imag_d(i) {}
    double real() const { return real_d; }
    duoble imag() const { return imag_d; }
    void real(double r) { real_d = r; }
    void imag(double i) { imag_d = i; }

    // 此处忽略细节
};
```

一旦接口中指定可以通过 Complex 对象得到浮点型的实部和虚部之后，如何在对象中存储这些浮点数就变成了类的私有实现细节。用户代码要想取得这些数据，就必须调用类的成员函数，而不是直接去访问数据成员。

```
Complex c(3.0, 4.0);
double d = c.real();
c.imag(0.0);
```

现在对于用户来说，惟一的不同就是他们为了获取值将不得不多输入两个字符（即括号）。由于 Complex::real 是一个内嵌函数，所以这两种形式产生的代码应该是一致的——将取值操作封装在一个内嵌函数中不会带来任何效率上的损失。同样，我们也可以将修改值的操作封装到内嵌函数中去。

现在，我们的类中再也没有公用数据了，为了将信息以其他格式存储而修改类的实现也变得容易起来：

```
// 文件 Complex.h
class Complex {
private:
    double r;
    double theta;
public:
    Complex(double re, double im); // 不再是内嵌函数了
    double real() const { return r*sin(theta); }
    double imag() const { return r*cos(theta); }
    void real(double); // 不再是内嵌函数了
    void imag(double); // 不再是内嵌函数了

    // 此处忽略细节
};

// 文件 Complex.c
Complex::Complex(double re, double im)
    : r(sqrt( re*re + im*im )),
      theta(atan2(im,re)) {
}

// "real(double)"和"imag(double)"留给读者作为练习
```

此时，用户代码必须重新编译，程序的性能也会有一定的影响，但程序的接口没有改变。原来可以正常工作的程序仍然可以正常工作。

我们也可以将实现改为把信息存储到其他的位置上去：

```
class Complex_rep {
private:
    friend class Complex;
    double real_d;
    double imag_d;
    Complex_rep(double r, double i): real_d(r), imag(i) {}
};

class Complex {
private:
    Complex_rep* rep;
public:
    Complex(double r, double i)
    : rep(new Complex_rep(r,i)) {}
    ~Complex() { delete rep; }
```

```
    // 此处忽略细节
};
```

我们将在第 3 章中解释为什么这么做。

2.3.1 表示不变量（Representation invariant）

公用数据还会使得我们的类很难以用来保证表示不变量。表示不变量是一个谓词，对于某个完全被构建好的对象，它都将得到真值。

例如，假设我们有一个用来表示有理数的类 Rational，它里面有两个整型数（一个用于分子，一个用于分母）：

```
class Rational {
private:
    int num_d;
    int denom_d;
public:
    Rational(int n, int d);
    int num() const { return num_d; }
    int denom() const { return denom_d; }
    voie num(int n) { num_d = n; }
    void denom(int); // 改变分母
};
```

如果我们的类具有这么一个表示不变量（分母 denom_d 不可能为 0），这将会使得我们的算法变得更简单，并且更高效。我们在每个可能改变分母的函数中都对新的分母进行检测，如果它是 0 的话，我们就向外报告一个错误[2]：

```
static void
check_zero(int d) {
    if (d == 0) {
        cerr << "Zero denominator in Rational\n";
        abort();
    }
}

Rational::Rational(int n, int d)
: num_d(n),
  denom_d(d) {
    check_zero(d);
}

void Rational::denom(int d) {
    check_zero(d);
```

2 如果我们使用的编译器支持异常，那么它应该抛出一个异常（详见第 10 章），而不是退出函数。

```
    denom_d = d;
}
```

有了这两个函数我们就可以保证不变量的成立，因此在 Rational 的其他成员函数中，我们并不需要担心出现分母为 0 的情况。如果我们将分母声明为一个公用数据成员，我们就无法来保证不变量的成立，用户代码也就可以在任意时刻将分母赋值为 0。有关 Rational 的所有操作也都将不得不对这种可能性进行检测。

综上所述，我们得到如下的结论：避免公用数据成员的出现。公用成员将使得我们很难去更改类的实现（如改变信息的存储格式或者存储位置）；它也无法保证表示不变量的长期有效性。

2.4 隐式类型转换

隐式类型转换（详情参见下面的回顾）会使得编译器能够在暗中修改我们所写的代码。由于这个原因，我们必须谨慎地提供和使用它们：人们很难理解那些有着大量隐式类型转换的代码。而且，使用这种转换的场合越多，我们就越可能得到有着歧义的函数调用，因此也就需要更多的显式转换来消除这种歧义。在本节中，我们将向读者展示一些经常发现在使用隐式类型转换的代码中的问题。我们同样也会向读者给出一些关于“该在何时使用隐式类型转换”的建议。

回顾：隐式类型转换

我们可以通过如下两种方式来声明一个从类型 F 到类型 T 的隐式转换：

- 在 T 中声明一个仅接受一个类型为 F 的参数的构造函数（这其中也包括那些使用了缺省参数值的构造函数，如：T::T(F, int = 0)）
- 在 F 中声明一个 operator T 的转换函数

如果在函数调用中无法找到形式参数与实际参数类型完全匹配的函数声明，编译器就将试图通过在实参上进行隐式转换来找到合适的函数。对于作用域中所有同名的函数，编译器会试着将用户自定义的(至多一个)隐式转换作用到每个实参之上来寻找匹配程度最高的函数声明。如果编译器找到这样的一个函数，那么它就会在目标代码中使用这个转换，最后调用该函数；否则，编译器就会向外报告一个编译错误。

```
class Rational {
public:
// ...
    Rational(int,int);
    operator double();
};
```

```
double sqrt(double);

main() {
    Rational r(4,1);
    double sq = sqrt(r);
};
```

在上面的代码中，r就被隐式地转换为一个double值，就好比如下的用户代码：

```
double dval = r.operator double();
dobule sq = sqrt(dval);
```

2.4.1 带有单个参数的构造函数

带有单个参数的构造函数同时也是一个隐式转换，这个事实让人觉得很不方便，因为我们希望的是该构造函数应该用于明确的对象创建过程中。只有在概念上将同样的信息由一种格式转换成另外一种格式的前提下，我们才允许有这样的隐式转换。如果在概念上，被转换后的对象与构造函数的参数不同，这就很让人费解。我们将通过一些例子来向大家展示这一灰色区域：

```
class String {
// 此处忽略细节
public:
    String(const char* = "");
};

void print_heading(const String&);
// ...
print_heading("Annual Report");
```

在调用print_heading时，我们就会得到一个隐式的从char*到String的转换；由于String和char*在概念上来说是同一抽象模型的两个不同表示方法，所以这样的操作是合理的。

```
class Rational {
// 此处忽略细节
public:
    Rational(long num = 0, long denom = 1);
};

int operator==(const Rational&, const Rational&);

int nonzero(const Rational& r) {
    return r == 0;
}
```

上面这个例子虽然不是很清晰，但同样也可以让人明了。0 在这被隐式转换成一个 Rational（在 Rational 的构造函数中，第二个参数有着一个缺省的值 1）。然而，在这样写的代码中，那个比较操作看起来要比它实际上的操作要显得更高效一些。它的显式版本如下：

```
int nonzero(const Rational& r) {
    return r == Rational(0,1);
}
```

上面的代码就可以给我们一个更好的关于该函数的实际性能如何的描述，并且它还可以向我们暗示出如下的优化：

```
int nonzero(const Rational& r) {
    static const Rational zero(0,1);
    return r == zero;
}
```

这使得我们可以节省下每次都创建和摧毁一个新的 Rational 对象 zero 所产生的开销。只有那些精明的程序员才可以从函数的最初版本中发现这种优化。

下面是另外一个例子：

```
#include <stdlib.h>
// 随机数产生器：
class Random_gen {
// 此处忽略细节
public:
Random_gen(long seed);
};

void play_game(Random_gen);

main(int argc, char* argv[]) {
if (argc > 1)
    play_game(atoi(argv[1]));  // 让人迷惑的地方
}
```

上面的程序中本来是不应该出现隐式转换的。一个用于产生随机数的对象和它用来产生随机数的种子（seed）应该分属于不同的对象；尤其是当它使用了一个 long 作为其构造函数参数时，它的类型就和这个种子完全不一样了。在上面的代码中，“传递给 play_game 的参数并不是 atoi 函数的输出”这个事实很容易被人所忽略。这段代码应该使用显式调用构造函数的方式来重写：

```
int main(int argc, char* argv[]) {
if (argc > 1) {
    Random_gen gen(atoi(argv[1]));
    play_game(gen); // 现在好多了
```

```
}
// ...
}
```

上面的代码和最初的代码在执行时是一样快的（不管我们有没有写构造函数的调用代码，编译器都会为我们产生它），但它更能体现程序的意图。

2.4.2 类型转换操作符

有着多个类型转换操作符的类在被使用时更容易产生编译时期的二义性问题。例如，有一个只带有一个类型转换操作符的类如下：

```
class String {
    char* rep;
public:
    String(const char* = "");
    operator const char*() const { return rep; }
    // 此处忽略细节
};

main() {
    String s("hello world");
    cout << s << endl;
}
```

上面的代码可以正常工作；由于我们没有声明 String 的输出操作符，代码中的 String 对象就被转换成一个 const char*用于输出。

如果我们往 String 中再添加一个类型转换操作符，会有什么现象出现呢：

```
class String {
    char* rep;
public:
    String(const char* = "");
    operator const char*() const { return rep; }
    operator int() const { return atoi(rep); } // 新增的转换操作符
    // 此处忽略细节
};
```

看上去这个新增的转换操作符会给我们带来很多的便利，因为它可以将 String 用于那些需要 int 的场合中；但是它却会破坏我们已有的代码：

```
main() {
    String s("hello world");
    cout << s << endl;  // 编译期错误:
                        // 无法确定进行哪种转换
}
```

现在对于 String 的输出，编译器的解析步骤将变成在“将 String 转换成 const char*再输出”和“将 String 转换为 int 再输出”这两种行为中挑选一种，因此我们就得到了一个编译错误。为了让上述代码顺利通过编译，我们不得不使用显式转换：

```
cout << (const char*)s;
```

在 String 中同时提供两种隐式转换使得程序中出现歧义的可能性大为增加。

不过这并不意味着我们不能将 String 转换成为 int，我们所要做的只不过是不再使用隐式的方式进行这种转换而已：

```
class String {
    char* rep;
public:
    int as_int() { return atoi(rep); }
    // 此处忽略细节
};
```

现在，如果希望进行从 String 到 int 的转换，我们就需要明确地调用 as_int 函数：

```
void process_key_value(const String& key, const String& val) {
    if (numeric_key(key)) {
        int value = val.as_int();
    // ...
```

这并不是个坏消息，使用明确的函数调用会使得程序更容易被人所理解。

2.4.3 内建类型之间的隐式转换

对于涉及到自定义类型的隐式转换，它们的规则比较固定也容易让人理解。但对于在内建的整型类型之间的隐式转换，它们的规则就显得令人难以捉摸，并且在语言的发展过程中，这些规则也有着一些变化。例如：

```
void f(long);
void f(int);
main() {
    f('x'); // 此处调用的是哪个 f?
}
```

上面的代码中调用的是 f(int)，但我们再来看如下的代码：

```
void f(long);
void f(unsigned int);
main() {
    f('x'); // 此处调用的是哪个 f?
}
```

它会产生一个编译时期的错误（有着歧义的函数调用）。可能你对于语言规则十分了解，

知道为什么会有这种现象发生，但那些将来维护这段代码的人可能就不清楚它发生的原因了。我们应该避开语言中的那些含义模糊的特性，如果真的需要对不同的内建整型进行不同的操作，我们最好是为每个整型都提供一个参数类型和它完全匹配的函数：

```
void output(logn);
void output(unsigned long);
void output(int);
void output(unsigned int);
void output(short);
void output(unsigned short);
void output(char);
void output(unsigned char);
void output(signed char);
```

现在，我们就可以确信下面代码的意思不但明显，而且也不会发生改变：

```
unsigned short s;
output(s);
```

2.5 操作符重载：成员或非成员？

在 C++中，对于操作符的重载有着两种方法：将该操作符作为成员函数或者不作为成员函数。在本节中，我们将会对如何为一个特定的操作符选择合适的重载方式给出一些评判标准。

显而易见的是，如果我们正在为一个不属于我们的类重载一个操作符，这个操作符应该是一个非成员函数：如果类不属于我们，我们也就不能向其中增加成员了！在本节的剩余部分中假定了我们可以决定是否将操作符定义为一个成员函数。

对于成员和非成员的选择结果会影响到使用该操作符的代码，例如，作为成员函数的操作符可以使用 this 指针，并且也可以用非限定形式来使用类中的成员。此外，这个选择结果还会影响到用户眼中该操作符的行为。如果操作符被实现为一个成员函数，那么我们就无法对第一个（最左边的）操作符进行隐式类型转换。这种不同存在于用户眼中，但对于编码来说则并不明显，因此它也应该成为我们在成员和非成员之间进行选择的一个主要评判标准。

2.5.1 一元操作符

我们先来看一个有关一元操作符的例子。假设我们有一个类 Vector（向量）[3]，它创建自一个 Direction（方向）对象和一个用 double 描述的数值：

```
class Direction {
// 此处忽略细节
};
```

3 译注：此处的向量是几何中的向量，而不是像标准库中那样用于表达一个数组的向量。

```
class Vector {
public:
    Vector(const Direction&, double magnitude = 0.0);
};
```

由于 Vector 的构造函数中第二个参数有缺省值，我们就可以从一个 Direction 对象经过隐式转换得到一个 Vector 对象。

现在我们希望为 Vector 重载一个一元操作符。如果我们把它定义为非成员函数，那么我们也就可以将它作用到 Direction 上去（因为 Direction 可以隐式地转换为 Vector）：

```
Vector operator-(const Vector&);

main() {
    Direction d;
    Vector v = -d;  // 合法的: operator-(Vector(d))
}
```

如果 operator-是一个成员函数，那么隐式转换就将被禁止：

```
class Vector {
public:
    Vector(const Direction&, double magnitude = 0.0);
    Vector operator-() const;
};

main() {
    Direction d;
    Vector v = -d;  // 编译期错误
}
```

我个人认为禁止隐式转换是一件好事，如同我们在 2.4 节中所看到的，使用隐式转换会使得代码难以被维护和被理解。出于这个原因，我建议：在可能的情况下，最好将一元操作符实现为成员函数。

2.5.2 =, [], ()以及 ->

按照 C++语言定义来说，这四种操作符必须被实现成成员函数。任何将它们中的某个实现为非成员的尝试都会造成一个编译期的错误：

```
extern int operator=(Complex&, const Complex&);
// 编译期错误: operator=必须是成员函数
```

2.5.3 其他的二元操作符

对于其他的二元操作符来说，是否将它们实现为成员函数取决于我们是否需要对左操作数进行隐式转换。对于赋值类操作符（如+=）来说，我们希望的是禁止对左操作数进行隐式转

换：

```
Complex c;
c += 5;
```

我们很难想象 c 在+=的操作实施前就被隐式转换成为其他的东西。如果上面的代码顺利地通过了编译，在它运行后 c 的值将不会得到改变，这一点肯定会让人迷惑不解。如果我们将赋值类操作符定义为成员函数的话，我们就可以禁止这种情况的出现，并且还可以保证它们和 operator =（它必须是一个成员函数）具有一致的行为。

此外，对于那些非赋值类从操作符来说，禁止对左操作数进行隐式转换但却允许对右操作数进行隐式转换同样会让人困惑。例如：

```
class Complex {
// 此处忽略细节
public:
    Complex(double = 0.0, double = 0.0);
    Complex operator+(const Complex&) const;
};

main() {
    Complex c(1.0);
    Complex d = c + 1.0;    // OK
    Complex e = 1.0 + c;    // 编译期错误
}
```

注意：我们在 Complex 的构造函数中使用了有缺省值的参数。这使得我们可以从 double 通过隐式转换得到 Complex（这时，构造函数的第二个参数将会是缺省值 0.0）。上面代码中对于 operator+的第二次调用出错的原因在于：在当前作用域中，没有一个合适的声明对应于这次函数调用；编译器不会对成员函数的左操作数进行隐式转换。如果我们将 operator+声明为非成员函数，那么上面两种形式的调用都将顺利地通过编译。

当两个操作数都是操作的输入，并且没有一个会受到操作的影响，那么我们最好将该操作符定义为一个非成员函数。这可以确保在有隐式转换的条件下，这两个操作数都可以以同样的方式来进行操作。

图 2.1 中归纳了本节中对于操作符是否该定义为成员函数的建议：

操作符	建议
所有的一元操作符	成员
= () [] ->	必须是成员
+= -= /= *= ^= &= != %= >>= <<=	成员
所有其他的二元操作符	非成员

图 2.1 操作符重载：成员或非成员？

2.6 重载、缺省值以及省略符

C++中允许用户写出这样的代码：使用不同（个数或类型）的参数来调用（似乎是）同一个函数：

```
foo();
foo(0);
foo(x,y,z);
```

（当然，这些函数都应该对它们的参数有着同样的抽象行为。）在C++中有三种方法可以这么做：函数重载、使用有缺省值的参数以及使用省略符。对于在同一个函数名中同时使用这三种方法中的多种所出现的结果，C++语言规则有着明确的定义。不过它们比较难以掌握，并且还容易让人觉得惊奇。即使我们能够很好地理解它们，那些维护我们代码的人也可能不一定能够完全掌握它们。对于同一个名字来说，我们应该对它选择一种方案并且一直坚持使用这种方法。

省略符只应该用在类似于printf这样的需要无限（或近似无限）个重载版本的函数中。在使用了省略符的函数中并没有对那些不定参数的类型进行编译期的类型检测，在函数内部我们还必须使用定义于头文件stdarg.h中的充满魔幻色彩的宏[4]来获得那些不定参数。如果函数没有正确获得该参数，那么函数的返回值就将无法确定。

假设我们已经去除了省略符，在使用缺省值和重载之间的选择通常都很容易。使用缺省值可以让我们使用多种不同的方法来同一个函数；使用重载则适用于需要调用不同的函数这种场合之下。

如果这两者之间有一个是另外一个的简化方式，并且同一个实现可以用在这两种场合之下，那么我们最好选择使用带缺省值的方式。例如，我们有一个类String，它的构造函数带有一个const char*的参数，但是我们有希望它有一个缺省的构造函数。为了简便，我们可以使用一个带缺省值的参数：

```
class String {
// 此处忽略细节
public:
    String(const char* = "");
};

String s;   // 等同于"String s("");"
```

当多个事物在概念上相似，但实现形式又不一致时，我们应该选择使用重载：

```
void print(const Data&);
void print(const Location&);
```

4 译注：即va_start、va_list以及va_end。

```
void print(const Employee&);
```

上面三个 print 在概念上来说是一样的，但它们的实现方法不一样，所以我们必须对它们使用重载。

2.6.1 缺省的指针和引用参数

当我们为指针参数提供缺省值时，下面这个小的语法陷阱就很容易出现：

```
void f(const char*= "");
```

编译器会将*=当作一个单独的 token 来处理，这样就将造成一个语法错误。我们应该在*后面增加一个空格：

```
void f(const char* = "");
```

对于有着缺省值的引用对象来说，编译器也是同样处理的：

```
const String empty;
void f(const String&= empty);  // 语法错误
void f(const String& = empty); // OK
```

2.7 Const

许多 C++程序员在开始使用 const 时都是用它来定义一些常数；例如将：

```
// C版本:
#define BUFF_LENGTH 1024
int buffer[BUFF_LENGTH];
```

写成：

```
// C++版本:
const int BUFF_LENGTH = 1024;
int buffer[BUFF_LENGTH];
```

这样做可以获得的好处是：编译器（或者其他工具）可以知道 BUFF_LENGTH 的名字和类型。而且这样做不会给程序的执行时间和尺寸上带来什么额外的开销，如果我们不去使用 BUFF_LENGTH 的地址的话，编译器也就不需要为它申请专门的存储空间。

然而，const 能做的并不仅仅是定义常数那么简单。通过声明一个“指向常量的指针”，我们就可以确保该指针指向的对象不会被改变。在本节中，我们将向读者展示指向常量的指针（或者引用）是如何影响我们的代码的。

请记住一点：将某件事物声明为 const（或者是指向 const 的指针或者引用）会引起额外的编译期检测，但它并不会导致编译器产生额外的代码。

2.7.1 常量引用参数

在 C++中，函数参数是以值的方式进行传递的。这意味着在被调用的函数中，存在着一份实际参数的拷贝；我们在被调函数中对该拷贝进行的改变不会反映到调用函数中去。

当一个自定义类型的对象以传值的方式传递到函数中去时，通过调用该类型的复制构造函数，它将和函数的其他实参一样被复制。当函数返回时，编译器会产生一些代码（调用析构函数）来摧毁这份拷贝。这样的复制操作可能会造成额外的开销，尤其是当被调函数又将该对象传递给其他函数时更是如此。在这种情况下，每个最外层的调用都可能会产生该对象的好几份拷贝（以及相应的构造和析构代码）。

在大多数情况下，即便对象在概念上是以传值的方式传递给函数，被调函数可能也不需要有着它自己的对于该对象的独立拷贝，它可以提供一个常量引用来使用这个对象：

```
// 正确但运行缓慢的代码：
// 不必要的 Telephone_number 的拷贝：
void dial(Telephone_number tn) {
// 此处忽略细节
}

// 改善后的代码：使用了一个常量引用：
void dial(const Telephone_number& tn) {
// 此处忽略细节
}
```

上面的第二份代码的速度要快一些，因为它避免了对输入参数进行复制并在函数返回时摧毁该复制的操作。此处使用 const 阻止了被调函数无意中对调用函数中对象的值进行修改。

使用引用给用户带来了一些重要的语法甜头：他们不再需要去记忆哪些参数是以指针的方式传递给了函数，哪些参数是以值的方式传递给了函数。如果在概念上来说，参数是以值的方式传递给函数，我们就可以以上面的方式来写调用语句，而“参数是以引用的方式传递给函数”这个实现细节就被隐藏起来了。

2.7.2 常量参数和常量指针

不同的 C++程序员对于 const 的态度不一样。有些人把它看做是在编译期间寻找 bug 的一个重要工具；其他人则认为相对它的好处来说，const 带来的麻烦要更多一些，并因此不去使用它。如果我们正在编写将要被他人使用的代码，第二种态度就不适合我们：即使我们自己不使用 const，其他使用我们编写的类的人可能也会使用它，因此我们不得不将一些适当的事物声明为 const 以允许其他人的使用。

在那些接受指针参数的函数中，这种情况出现得最多。如果函数只是通过指针来读取（它并不会向被指向的对象存储或者更改它的内容），在函数声明时，我们应该将该参数声明为一

个指向常量的指针：

```
class String {
public:
    String(const char* = "");
// 此处忽略细节
};
```

上面的声明保证了 String 的构造函数不会通过它的指针参数进行存储活动。（任何试图在 String 的构造函数中对它进行的存储活动都将导致一个编译期错误。）如果我们把 String 的构造函数声明为带有一个类型为 char*而不是 const char*的参数，那么所有有着常量指针的用户就无法用这个构造函数来构建一个 String 对象：

```
// 在String.h中：
class String {
public:
    String(char* = ""); // 应该为"const char*"
// 此处忽略细节
};

// 在用户代码中：

main() {
    const char* hello = "hello world";
    String s(hello);    // 编译期错误：
                        // 找不到合适的 String 构造函数
}
```

同样的情况也适用于接受引用参数的函数：如果函数不会通过引用来存储内容，那么它应该接受一个常量引用作为其参数。常量引用参数也使得将一个指向未命名的临时对象的引用传递给函数成为了可能。

```
Thing get_a_thing();

void look_at_thing(const Thing&);
void change_thing(Thing&);

look_at_thing(get_a_thing());  // OK
change_thing(get_a_thing());   // 编译期错误
```

对于 change_thing 的调用会产生一个编译期的错误：将一个未命名的临时对象作为一个非常量指针传递给函数是非法的。如果在被调函数中对引用参数的值进行了修改，但调用函数却忽略了这种修改，我们认为这种行为是一种 bug；因此 C++中也就增添了这么一条规则来禁止这种 bug 的产生。如果我们真的想那么做的话，我们就必须得创建一个具名对象：

```
Thing t(get_a_thing());
change_thing(t);    // OK
```

声明一个指向常量的指针关注的是该指针，而不是该指针指向的空间。编译器并不能确保被指向的数据不会被改变；它能确保的是，数据不会是通过该指针被改变的。我们仍然可以使用其他方式来改变被指向的对象的值：

```
void do_callback(const int* ip, void(*callback)()) {
    cout << *ip << endl;
    (*callback)();
    cout << *ip << endl;
}
```

即使 ip 是一个指向 const int 的指针，我们仍然不能保证这两次打印的数字会是一样的。如：

```
int i = 5;
void bump_i() {
    ++i;
}

main() {
    do_callback(&i, bump_i);
}
```

它就将打印：

```
5
6
```

在两次打印之间，那个回调函数会修改 i 的值。

一个语法上的小缺点

当我们在 do_callback 中通过函数指针来调用那个回调函数时，我们先对函数指针进行解引用，然后再调用该函数：

```
(*callback)();
```

然而，如果我们直接通过()操作符来“调用”该函数，我们将得到和前面一样的结果：

```
callback();
```

这两种方式得到的代码将完全一样。

这只是一个语法上的甜头而已，我并不推荐大家这么做。当我们使用第一种形式时，我们可以很清楚地知道函数是通过函数指针来调用的；如果我们使用第二种形式，我们必须了解“函数”的类型实际上也就是“函数指针”的类型。相比而言，少输入几个字符所得来的好处还不值得我们去为它而生成难以理解的代码。

2.7.3 常量成员函数

那些不会修改对象值的函数应该被声明为 const（详情参见后面的“回顾”）。这使得其他

人可以使用我们编写的类来创建常量对象，并使用编译器来确保对这些对象所进行的成员函数调用不会修改它的值。

在未命名的临时对象上调用非常量成员函数

即使我们不能传递一个指向非常量未命名临时对象的引用，我们还是可以合法地在一个未命名的临时对象上调用一个非常量成员函数：

```
class String {
public:
    void capitalize(); // 非常量成员函数
};
// ...
String make_up_name();
make_up_name().capitalize();    // 可以但不应该这么调用
```

这种做法凸显了C++语言定义的一个失误，我期望ISO/ANSI C++标准委员会能够在随后的标准制定过程中把它给改正；不管如何，我们都不应该使用这种用法。如果我们希望从对象来调用一个非常量的成员函数，我们必须明确地定义这个对象并给它一个名字：

```
String name(make_up_name());
name.capitalize(); // 稍好的做法
```

这将保证该名字所代表的对象在程序离开当前定义它的语句块前都不会被摧毁。

回顾：常量成员函数

通过在函数的声明体和定义体的参数列表后面添加关键字 const，我们可以把一个成员函数声明为一个常量成员函数：

```
// 在String.h中
class String {
public:
    // 此处忽略细节
    int langth() const
    void capitalize(); // 非常量成员函数
};

// 在String.c中
int String::length() const {
// 此处忽略细节
}

void String::capitalize() { // 非常量成员函数
// 此处忽略细节
}
```

我们只能对常量对象调用常量成员函数：

```
const String("hello");
int len = s.length();   // OK
s.capitalize(); // 编译期错误：对常量对象调用非常量成员函数

String t("world"); // 非常量 String
len = t.length();   // OK
t.capitalize();     // OK
```

在常量成员函数的定义体中，对象的所有数据成员都是常量，“this”指针也是一个“指向常量对象的常量指针”，而不是“指向对象的常量指针”。

会改变对象状态的常量成员函数

C++的语言规则确保：除了明确地使用了类型转换，常量成员函数不会修改对象的状态（数据成员）。然而，某些在概念上为常量的操作可能也会改变对象中某些成员的值；对于这种情况，我们应该把它们作为实现细节向用户隐藏起来。通过使用这种方法，用户就可以在不清楚实现细节会修改对象中的某些私用数据成员的情况下，对一个常量对象进行某些在概念上来说不会更改对象状态的操作。

例如，假设我们正在使用的 Complex 类中存储有值的极坐标形式。我们可能需要将它的笛卡儿坐标的值也缓冲在对象中，以节约以后对它们重复计算所带来的运行时间。现在，每个对象都包含有一个布尔值，我们用它来判断缓冲值是否有效：

```
typedef unsigned char Boolean;
class Complex {
private:
    double r, theta;
    double real_cache, imag_cache;
    Boolean real_cache_valid;
    Boolean imag_cache_valid;
public:
    Complex(double real, double imag);
    double real_part() const;
    void real_part(double);

    double imag_part() const;
    void imag_part(double);
};
```

我们在构造函数中将缓冲值标志位设为 0，以表明缓冲值目前是无效的：

```
#include <math.h>
Complex::Complex(double re, double im)
: r(sqrt( er*re + im*im )),
  theta(atan2(im,re)),
```

```
  real_cache_valid(0),
  imag_cache_valid(0)
{}
```

在概念上来说，我们应该把那些用来获取（而不是设置）这些值的函数定义为常量成员函数，但是它们实际上也会修改对象中的缓冲值：

```
double Complex::real_part() const {
    if (!real_cache_valid) {
        (double&)real_cache = r*sin(theta);
        (Boolean&)real_cache_valid = 1;
    }
    return real_cache;
}
```

为了在一个常量成员函数中对数据成员进行修改，我们必须使用类型转换来去除该成员的常量性。在 C++中，这种做法不但合法，而且只要类中带有一个构造函数，它的行为就将正确无误。（如果类中没有构造函数，那么我们用来去除常量性的类型转换将得到未定义的结果；之所以有这样的规则是因为我们希望编译器能够将类似于 const int 这样的事物放置到 ROM 中去。）

2.8 返回值为引用

函数可以返回一个引用。返回引用和返回指针差不多；被引用的对象在函数返回时必须仍然存在。尤其是，我们不能返回一个自动变量（automatic variable）的引用，其原因和我们不能返回一个指向自动变量的指针一样：当函数返回时，所有的自动变量都将会被摧毁，这时我们所能得到的只是一个空悬（dangling）引用。

例如，由于被赋值的对象在赋值结束后仍然存在，我们的 String 类中的赋值操作符就可以返回一个指向*this 的引用。然而，用来创建一个新的 String 对象的连接操作的函数则必须返回一个值：

```
String  /* 不是 String& */
operator+(const String& left, const String& right) {
    String result;
    // 在此处构建 result
    return result;
}
```

由于在 operator+返回时，result 会被摧毁，所以我们不能返回一个指向 result 的引用。我们必须用值的方式返回它，这将产生 result 的一个拷贝，函数会把该拷贝返回给调用函数。

在此我们的程序会有一些性能上的损耗，因为我们必须先构建一份对象的局部拷贝，然后再通过调用复制构造函数把它给返回给调用函数；不过由于这种用法的普遍性，许多的编译器

都可以把这份额外的拷贝给优化掉[5]。

2.9 静态对象的构造

我们也可以用构造函数来对那些有着 static 存储类型的对象进行初始化：

```
static String hello("Hello world");
```

C++编译系统确保：所有的静态对象在它们被使用之前都会被初始化。很多的编译器实现的做法是将所有的静态对象放到 main 函数被调用之前进行初始化，但 C++语言规则中也允许我们在（动态的）链接时期用动态的链接器来初始化编译单元中的所有对象。

对于在同一个编译单元中出现的静态对象，它们的初始化顺序和它们在代码中出现的顺序是一致的：

```
String dafault_name("foo");
String dafault_src_file(default_name + ".c");
```

在上面的代码中，我们可以确保 default_name 的初始化操作要早于 default_src_file 的初始化操作。在同一文件中出现的静态对象，它们的析构顺序和它们的构造顺序正好相反，因此 default_src_file 必须早于 default_name 被析构。

在不同文件中的初始化操作的顺序则是未定义的：

```
extern Sring default_name;
String default_src_file(default_name + ".c");  // Bug
```

我们无法保证，default_name 会早于 default_src_file 被创建。

对于上面的那个问题，我们并没有一个简单的解决方法。如果我们有着一个依赖于定义在其他文件中的其他静态对象的静态对象（如 default_src_file），我们就应该试着将这两个对象放在同一个文件中（这样我们就可以得到它们的初始化的确切顺序）。如果我们做不到这点，我们就不得不将这些初始化操作延迟到 main 函数被执行时才执行它们：

```
extern Sring default_name;
String* default_src_file = 0;

static void init_names() {
    default_src_file = new String(default_name + ".c");
    // 其他的初始化操作
}

main() {
    init_name();
```

5 译注：即所谓的 NVO。

如果我们的代码来自于一个函数库，在那里并没有我们所期望的 main 函数，那么我们还可以在每个使用了具有依赖关系的静态对象的函数中对该对象进行检测，看看它是否已经被初始化了：

```
extern Sring default_name;
String* default_src_file = 0;

void routine_that_uses_name() {
    if (default_src_file == 0)
        default_src_file = new String(default_name + ".c");
    // 使用 default_src_file
```

我们应该避免在一个给定的头文件中初始化对象，因为它会造成每个包含该头文件的编译单元都拥有这份初始化代码。模板特性（见第 7 章）的一个副作用就是：它会增加给定程序中编译单元的个数（因为通常每个模板对会被实例化为一个单独的编译单元）。当使用模板时，我们很容易就能得到上百个编译单元的程序。对于这样的程序来说，要求同一个初始化在每个编译单元中都有着冗余的运行时开销的方式会显得太昂贵了。

2.10 小结

- 判断缺省的复制构造函数和赋值操作符的行为是否符合我们的期望，必要时重新实现它们。
- 避免使用赋值操作来初始化成员；使用构造函数初始化语法来完成初始化操作。
- 当编写赋值操作符时，请检测 s = s 这种情况。
- 避免出现公用数据。
- 尽可能少地声明和使用隐式类型转换。避免在同一个类中出现两个（或多个）转换操作符。
- 一元操作符、赋值类操作符、()、[]、以及->应该被定义成成员函数，其他操作符应该被定义为非成员函数。
- 使用缺省值来为同一个函数提供不同的调用方式；使用函数重载来为同一个抽象操作提供多种实现。
- 除非被调用函数需要它自己的对象拷贝，否则请使用传递常量引用的方式来调用函数。
- 在可能的情况下，将指针参数声明为指向常量的指针，将引用参数声明为指向常量的引用。
- 在可能的情况下，将成员函数声明为 const。
- 不同文件中的静态数据的初始化顺序没有被明确定义。

2.11 问题

1. 复制构造函数 X::X(X)在 C++中是非法的。请解释其原因。

2. 我们的 String 类的构造函数通过调用

```
new char[strlen(cp)+1];
```

来为它的数据获取存储空间。不过在字符串中只有 strlen(cp)个字符，那么多出来的那个字符是做什么用的呢？

3. 我们的 File 类假设所有的 File 对象都表示一个惟一的文件描述符——对一个 File 进行复制会产生一个新的描述符。假设我们希望复制一个 File 对象时只产生一个和该对象共享已有描述符的对象——也就是说这个新的对象不会对已有描述符的数目进行递增。我们应该如何修改 File 类来达到这个目的呢？（提示：仅仅不让复制构造函数去修改那个计数器是不够的。）

4. 一个编译器制造商可能是为了节省开发成本，没有去实现内嵌的展开：也就是说，所有的对于内嵌函数的调用都会被解释为一次真正的函数调用。对于 C++来说，这样做是合法的，但对于使用该编译器的用户来说，这样做可能会是一场灾难。为什么？

5. 有个程序员可能希望为 double 和 Rational 的任意组合提供一个加法操作符。请为他提出两种解决方案，并对它们的优缺点做出评价。

6. 向上题中的加法操作增添对 Complex 的支持并重复上题中的讨论。

7. 为我们已有的 Complex 类（使用了极坐标表示）实现设置其（笛卡儿坐标系中的）实部和虚部的函数。

8. 我们的 Complex 类中用来设置其实部和虚部的函数的返回值是 void。我们可以把它们的返回值改成什么样才有意义呢？是否存在着其他的答案可以使得问题更加清楚？如果没有的话，你会建议使用那种方法呢？

9. 假设我们正在使用 String 类，它提供了一个 operator const char*的转换函数：

```
class String {
public:
    operator const char*() const;
    // 此处忽略细节
};
```

再假设我们有一个函数 remove_blanks，它带有一个类型为 char*的参数，我们用它来去除它的参数所指向的字符串中的空格：

```
void
remove_blanks(char* cp) {
    char* p = cp;
    while(*p) {
        if (*p != ' ')
```

```
            *cp++ = *p;
        ++p;
    }
    *cp = '\0';
}
```

最后假设有个用户，他试图用这个函数来去除掉一个 String 对象中的所有空格，为的是将这个 String 对象变短，而不是变长：

```
String s(" hello world ");
remove_blanks((char*)(const char*)s);
```

请问他这样做安全吗？为什么？

10. 下面的类中有什么错误？

```
class Node {
// 此处忽略细节
public:
    Node(); // 叶节点
    Node(const Node&); //一元
    Node(const Node&, const Node&);// 三元
};
```

我们应该如何更改它的接口来修正这个问题？

第 3 章

句　柄

数据抽象给我们带来的一个主要好处就是：对象的物理结构没有必要反映出它的抽象结构来。我们展示给用户看的模型具有的某些物理属性也不一定需要以模型中的形式出现在实现中。只要我们的对象可以和抽象模型相匹配，就可以以我们喜欢的方式来实现它。

尤其是当抽象模型表示的是一个对象“包含”另外一个对象时，我们的实现也不需要将被包含的对象作为一个数据成员放置到包含它的对象中去，而可以让包含对象拥有一个用来表示被包含对象的句柄（handle）。（我们通常把这个用来实现被包含对象的对象称为 rep。）从概念上来说，句柄是一个指向某种 C++对象的指针，它通常也就是一个普通的 C++指针。我们使用术语“句柄”的原因是：句柄本身也可以是某种 C++对象，它可以用来提供和指针一致的功能；我们将在第 7 章中对此给出一些例子。

句柄可以给我们带来如下的好处：

- 我们可以在实现中用尺寸大小固定的（constant-sized）对象来表示尺寸大小不定的（variable-sized）值。
- 我们可以在实现中用运行时绑定（run-time bounding）而不是编译时（compile-time bounding）绑定的方式来处理对象。
- 对于实现的改变通常只会引起一次重新链接，而不是重新编译。
- 我们可以对他人隐藏对象的实现。

不过有利必有弊，句柄的实现需要我们编写更多的代码，并且它还会带来一些运行时的开销。

在本章中，我们将会展示几种使用句柄的方法。首先，我们来看一些大家都很熟悉的例子。

3.1　一个 String 类

如果我们用 sizeof 操作符去查看 C++中对象的大小，总是可以得到一个固定的值。不过，我们并不需要保证值的表示形式也具有同样的大小；我们可以让对象包含有一个（有着固定大小的）句柄用以指定一个（具有不定大小的）值。当对象被创建时，构造函数会使用 new 操作符来为该值获得必要的空间。

我们的 String 类中的 rep 是一个指向堆中以‘\0’结尾的字符串。每个 String 对象都会包含一个类型为 char*的句柄：

```
class String {
private:
    char* rep;
public:
    String(const char* = "");
    ~String();
    String(const String&);
    const String& operator=(const String&);
};
```

所有包含了以指针为实现的句柄的类都应该至少包含如下的函数。

3.1.1 构造函数和析构函数

包含对象的构造函数会初始化指向 rep 的句柄。对于我们的 String 类来说，它的构造函数会创建该 rep，并在初始化时将指针指向它：

```
String::String(const char* c)
: rep(new char[strlen(c)+1]) {
    strcpy(rep,c);
}
```

String 的析构函数则负责删除 rep：

```
String::~String() {
    delete [] rep;
}
```

3.1.2 复制构造函数

缺省的复制构造函数会复制句柄，而不是 rep。如果句柄是一个普通的指针，由于复制会最终导致两个 String 对象指向同一个 rep，这将使得程序出错。当其中的一个 String 对象被析构时，它所拥有的 rep 也会被同时释放，这将使得另外一个 String 对象指向的 rep 成为一块已经被释放了的内存。

为了避免这种情况的出现，在我们的 String 类中，复制构造函数会创建一个新的 rep：

```
String::String(const String& s)
: rep(new char[strlen(s.rep)+1]) {
    strcpy(rep,s.rep);
}
```

3.1.3 赋值操作符

当缺省的复制构造函数有问题时，缺省的赋值操作符也可能因为同样的原因变得有问题。

而且，在赋值操作符中，我们还需要在复制 rep 前将已有的 rep 释放掉：

```
const String& String::operator=(cosnt String& s) {
    if (rep != s.rep) {
        delete [] rep;
        rep = new char[strlen(s.rep)+1];
        strcpy(rep,s.rep);
    }
    return *this;
}
```

由于现在每个 String 的对象都拥有同样的大小，我们就可以声明一个 String 的数组了；至于它们的值的大小不一样这个事实则被隐藏于我们的实现细节之下了：

```
main() {
    String t("C++");
    String s("ADD ONE TO COBOL");
    String array[10];
    array[0] = s;
    array[1] = t;
}
```

由于包含字符的内存空间并不是 String 对象的一部分，所以它并不会影响到 sizeof(String) 的结果。

3.2 使用计数器来避免多份拷贝

在我们的 String 类中，每次当我们复制 String 时，我们都会将 rep 也复制一份。其实我们也可以通过避免这些复制操作来节省一些程序的运行时间；我们的解决方法就是让多个句柄同时指向一个 rep。这样，在复制一个 String 对象时，我们可以只复制这个句柄而不是 rep，这时的句柄变成了一个用来表示“指向 rep 的句柄数”的计数器。当指向它的句柄数目为 0 时，我们将删除这个 rep。

对于我们的 String 类来说，我们使用了一个新的类 String_rep 来封装指向数据的指针和那个计数器：

```
class String {
private:
    class String_rep {
    public:
        int use_count;
        char*   chars;
            String_rep(const char*);
            String_rep(const String_rep&);
            ~String_rep();
```

```
        void    increment();
        void    decerment();
        const   String_rep& operator=(const String_rep&);
    };
    String_rep* rep;
public:
    String(const char* = "");
    ~String();
    String(const String&);
    const String& operator=(const String&);
};
```

此处 String_rep 是一个嵌套类：它声明于 String 类中。由于我们将它声明在 String 的私用段中，所以它只能被 String 的成员函数所访问。

注意：有些 C++编译器可以不支持嵌套类，或者是对嵌套类的支持有问题。如果我们手头的编译器无法正确处理嵌套类，我们也可以通过另外一种方法来实现同样的效果：将 String_rep 声明为一个非嵌套类，它里面的所有成员（包括构造函数）都被声明为私用的，然后将 String 类声明为它的友类：

```
class String_rep {
private:
    friend class String;
    int use_count;
    char*   chars;
        String_rep(const char*);
        String_rep(const String_rep&);
        ~String_rep();
    void    increment();
    void    decerment();
    const   String_rep& operator=(const String_rep&);
};
```

这同样可以保证用户无法构造 String_rep 的对象，但它带来的弊端就是：我们不得不把 String_rep 放置到全局的名字空间（参见 9.8 节）中去。

3.2.1 String_rep 的实现

String_rep 的构造函数很简单：它会为字符串申请空间，将字符串复制到该空间中去，然后把计数器置为 0。它的析构函数会释放掉构造函数中所申请的内存空间：

```
String::String_rep::String_rep(const char* cp)
: use_count(0),
  chars(new char[strlen(cp)+1]) {
    strcpy(chars,cp);
}
```

```
String::String_rep::~String_rep() {
    delete [] chars;
}
```

String_rep 中还有着其他两个会被 String 使用的成员函数。当 String 开始使用 rep 时，它会调用 increment()；当 String 用完 rep 后，它就会调用 decrement()。

```
void String::String_rep::increment() {
    ++use_count;
}

void String::String_rep::decrement() {
    if (--use_count == 0)
        delete [] this;
}
```

由于这两个函数很小，而且也会被经常使用，我们最好将它们声明为内嵌函数。

在 decrement 中，当计数器被清 0 时，我们会释放掉 String_rep。对于诸如 decrememnt 这样包含如下语句的函数：

```
delete this;
```

我们应该谨慎对待，因为这种“自杀”性质的成员函数会摧毁调用它们的对象。要想让它正常工作，被删除的对象必须存在于堆中。对于 String_rep 来说，它总可以得到正确的结果，因为我们在 String 的构造函数中会将所有的 String_rep 都构造在堆中——即使我们的 String 对象本身是一个自动（或者静态）对象。在使用了自杀性质的成员函数后，我们必须确保在对象被摧毁后该对象的成员不再被使用。

虽然我们的 String 类不会复制 String_rep，我们同样也仔细地对 String_rep 的缺省复制构造函数和缺省赋值操作符进行了考虑。如果它们不能达到我们的期望，而我们又对此无动于衷，那么对我们用来复制和修改 String_rep 的库的修改将会导致 bug 的产生（译见 2.1.2 节）。正如前面所描述的那样，由于它们只是简单地复制了字符串指针而不是字符串，所以这些缺省的行为不符合我们的预期。我们也可以通过把它们声明为私用以利于我们亲自实现它们。

这两个函数都会把 chars 所指向的字符串给复制一份。更加有意思的是 use_count 这个数据成员；我们用它来指示到底有多少个 String 对象指向该对象。当一个对象中包含有类似于 use_count 这样（用于指示对象信息，而不是对象的值）的数据成员时，我们通常需要把缺省的复制构造函数和赋值操作符都重新实现，不让它们简单地复制该数据成员。复制构造函数应该将该成员初始化为一个包含有正确值的新对象，赋值操作符则不应该改变该成员：

```
String::String_rep::String_rep(const String::String_rep& orig)
: use_count(0),
  chars(new char[strlen(orig.chars)+1]) {
    strcpy(chars,orig.chars);
```

```
}

const String::String_rep&
String::String_rep::operator=( const String::String_rep& orig) {
    if (this != &orig) {
        delete [] chars;
        chars = new char[strlen(orig.chars)+1];
        strcpy(chars,orig.chars);
    }
    // use_count 没有被改变
    return *this;
}
```

在 String_rep 的复制构造函数中，我们把 use_count 初始化为 0；在它的赋值操作符中，我们并没有改变该计数器。

3.2.2 String 的实现

在 String 类中，我们必须维护 String_rep 中的使用计数器。这包括两部分：在创建 String 对象时调用 increment；以及在摧毁 String 时调用 decrement：

```
String::String(cosnt char* c)
: rep(new String_rep(c)) {
    rep->increment();
}

String::~String() {
    rep->decrement();
} .

String::String(const String& s)
: rep(s.rep) {
    rep->increment();
}
```

任何改变 String::rep 值的操作都必须对计数器进行同时更新：

```
const String& String::operator=(const String& s) {
    if (rep != s.rep) {
        rep->decrement();
        rep = s.rep;
        rep->increment();
    }
}
```

再一次故调重弹，我们必须在使用赋值操作符时仔细地检测 s = s 这种情况，以避免该操作中调用的 decrement 会将 rep 删除。

注意：在我们的 String 中并没有明确地对 String_rep 进行删除，它只是对 rep 进行递减操作。当计数器归 0 时，String_rep::decrement 会自动将 rep 摧毁。

3.2.3 写时复制（Copy on write）

两个或多个 String 共享同一个 rep 的事实只是我们实现中的细节；我们必须确保所有会导致 String 值改变的操作不会导致其他共享该 rep 的 String 对象的值也发生改变。我们对于这种操作的解决办法通常都是：（虽然当原有的计数器值为 1 时，该操作还可以被优化掉）在这种操作中分配一个新的 rep，让它来维护被更改后的值。例如，有这么一个成员函数，它会强制将 String 中的所有字符都改成小写的形态：

```
#include <ctype.h>
void String::to_lower() {
    if (rep->use_count > 1) {
        String_rep* new_rep = new String_rep(rep->chars);
        rep->decrement();
        rep = new_rep;
        rep->increment();
    }
    assert(rep->use_count == 1);
    for (char* cp = rep->chars; *cp; ++cp)
        *cp = tolower(*cp);
}
```

该函数可以确保它里面的 rep 在被转化成小写之前没有被其他对象所共享。

3.2.4 一个告诫

使用了计数器的类要比没有使用计数器但功能相同的类要复杂得多，并且所有围绕着该计数器的操作都将导致程序执行时间的显著增加。如果用来复制值的时间足够短（这可能是由于该值占据的空间很小，导致复制它的成本很低；也可能是因为我们很少去复制它们），将类改为使用计数器的方式会导致程序的运行变慢。我们应该经常做一些性能上的度量，以确信这种优化的结果不会让我们太失望。

作为一个例子，我们可以比较使用了计数器的 String 类和没有使用计数器的 String 类。我用了一个测试程序来创建 1000 个空的 String 对象，并对每个 String 对象赋值了 0 到 3 次，最后再把它摧毁。

由于使用计数器的好处来自于我们不需要复制 String 中的字符串，我们可以预期：当字符串很大并且它们的值被经常复制时，使用了计数器的那个类应该会在比较中占优。在我的计算机上也确实是这样的。如果我们一直不去对 String 赋值，而只是对它进行创建和摧毁，那么使用了计数器的那个类将会在比较中落败：对使用计数器的 String 进行创建和摧毁的操作要比没有使用计数器的那个 String 慢大概 80%。如果我们只对每个字符串进行一次复制，这两个版

本所得到的比较结果将非常接近，只是在字符串包含的字符小于 25 个时，没有使用计数器的那个才略占上风。如果某个字符串被复制了两到三次，那么就一直是使用了计数器的版本要快一些。于是，我们得到了结论：在本例中，使用了计数器的版本在某些程序（其中对 String 的平均赋值超过一次）中要快一些，在其他程序中则要慢一些。

对于那些用来管理计数器的代码，我们应该仔细地对之进行审察。那些牵涉到计数器使用的 bug 通常都是很难找到的。

3.3 避免进行重编译：Cheshire Cat

如果我们可以很好地隐藏类的实现，那么对于实现的改动将不会导致已有代码的崩溃，不过它通常都会要求我们重新编译这些代码。例如，对于内嵌函数定义的改动，对于对象大小的改动，以及对于使用内嵌函数访问的对象结构的改动都将导致程序的重新编译。我们可以使用句柄来避免这些重编译：如果我们改动了 rep 的实现，用户要做的只是重新链接（而不是编译）一下他们的程序而已。不过，这也会给我们带来一些运行时的开销。

Carolan 描述了这样一种方案（他把它称为“Cheshire Cat”）：在程序中同时维护两个不同的头文件。在我们提交给用户使用的那个公用头文件中，我们只是告诉用户我们定义了 rep 这个类，并且在对象中用一个句柄来指向它。下面是一个新版本的 String 类，在其中我们没有使用计数器，而是使用了句柄来减少编译时期的依赖关系：

```
// 文件 String.h
class String_rep;

class String {
private:
    String_rep* rep;
public:
    String(const char* = "");
    ~String();
    String(const String&);
    const String& operator=(const String&);
};
```

这个文件提供的几乎是一个纯粹的接口——它所暴露出的惟一实现细节就是：我们使用了一个 rep。在该头文件中，我们不可能将使用了 String_rep 中成员的函数定义为内嵌函数，因为我们并没有提供 String_rep 的声明。

另外一个头文件是一个私有的头文件，我们不需要将它分发给我们的用户。它里面包括了对 rep 类的完整声明：

```
// 文件 String_priv.h
```

```
#include "String.h"
class String_rep {
private:
    friend class String;
    int use_count;
    char*   chars;
        String_rep(const char*);
        String_rep(const String_rep&);
        ~String_rep();
    void    increment();
    void    decrement();
    const String_rep& operator=(const String_rep&);
};
```

我们使用该私有头文件来编译 String 和 String_rep 的成员：

```
#include "String_priv.h"
String::String(const char* cp)
: rep(new String_rep(cp)) {
    rep->increment();
}

/* String 和 String_rep 其他成员函数的定义 ... */
```

注意：即使 String 中的成员函数不能是内嵌的，String_rep 中的也可以，这是因为所有涉及到 String_rep 成员的函数都可以对 String_rep 的定义进行完全的访问。

如果 String_rep 的实现被更改了，但 String 的却没有，我们的用户则没有必要重新编译他们的程序：他们只需要将程序重新链接一遍就可以了。这是因为他们的 String 对象结构并没有改变，并且也没有任何内嵌的 String 成员函数知道 String_rep 的结构。String 实现中那些了解 String_rep 结构的代码早已经被重新编译过了。

除了每次操作时的额外的内存引用，以及不能让涉及到 rep 的成员函数成为内嵌函数之外，我们都可以确信：以后对于实现的更改（而不是接口的更改）都不会导致重新编译的产生，它们只会要求我们对程序进行重新链接而已。

这样做值得吗？我们必须针对不同的情况来做出我们的判断。当函数比较小而且使用很频繁时，由额外的内存引用带来的运行时开销以及无法将函数内嵌化这两点都将对程序带来重大的影响。

3.4 使用句柄来隐藏设计

某“无名氏”[1]说过：

[1] 但我们最好还是认同他的言论。

朋友就是那种可以接触到你私有空间的人。

这话不错，但除非我们明确地进行阻止，否则任何人都可以窥探到我们的私有空间。

我们说：C++“隐藏”了实现细节，使得通过编写依赖于类中私用部分的程序变得不可能。但我们并没有方法阻止其他人从头文件中的声明得到实现的部分细节。

作为类的提供者，我们可能甚至希望用户不能阅读到我们的实现细节。例如，我们的类可能是围绕着一个我们不希望公开的私有算法来设计的。如果我们使用了“Cheshire Cat”这种方法，不把私有的头文件分发出去，我们的用户就无法获得任何有关我们实现的东西。我们应该确保，自己所分发出去的二进制文件中不会包含有任何有关 rep 的调试信息！

3.5 多种实现

当对象使用了句柄时，构造函数可以从几种不同的实现中来选择一种用以构建 rep。例如，我们可能会有两个不同的用于 String 中的 rep 的实现：其中的一种速度比较快，但它只能处理那些比较短的字符串；另外一种要稍慢一些，但它可以处理任意长度的字符串。我们就可以在 String 的构造函数中通过对字符串长度的判断来选择合适的实现。

我们可以通过将所有的实现派生自一个公有的基类来达到这个目的。对于我们的 String 类来说，String_rep 就是那个公有的基类，它指定了其他实现必须提供的操作。我们用派生类 Small_rep 和 Big_rep 来表示这两种不同的实现。每个 String 对象都将包含有一个 String_rep* 的句柄，它被用来指向一个 Small_rep 或是一个 Big_rep。所有对 rep 的访问都是通过对句柄调用虚拟函数来完成的：

```
class String {
private:

    class String_rep {
    public:
        virtual String_rep* copy() const = 0;
        virtual ~String_rep() {}
        virtual char operator[](int) const = 0;
        // 其他操作被忽略 ...
    };

    class Small_rep : public String_rep {
    public:
        Small_rep(const char*);
        ~Small_rep();
        String_rep* copy() const;
        char operator[](int) const;
    };
```

```
    class Big_rep : public String_rep {
    public:
        Big_rep(const char*);
        ~Big_rep();
        String_rep* copy() const;
        char operator[](int) const;
    };

    String_rep* rep;

public:
    String(const char* = "");
    String(const String&);
    ~String();
    const String& operator=(const String&);
    char operator[](int) const;
    // 其他操作被忽略
};
```

在 String_rep::copy 和 String_rep::operator[]声明中的“=0”表示这两个函数是纯虚函数，它将在派生类中被重写。我们将在 4.6 节中对纯虚函数进行进一步的讲解。

String_rep 中有一个虚的析构函数，但没有构造函数。虚析构函数的惟一用途就是强制让所有派生类中的析构函数都成为虚函数；这可以确保对于 String_rep*的删除操作会调用到正确的派生类中的析构函数。

每个 String 对象都应该能够对它的 rep 进行复制（例如，实现复制构造函数）；由于 String 对象不需要了解它所使用的是哪种 rep，我们应该用一个虚函数来实现复制操作。String_rep::copy()就是用来复制调用它的对象并返回一个指向复制后得到的拷贝的指针。

函数 operator[]只是作为一个用来描述“应该被 rep 实现的 String 操作”的例子出现在这里。

3.5.1 我们的 String 类

基于上面的讨论，String 中惟一了解不同 rep 之间的区别的成员就是它的某个构造函数：

```
const int STRING_THRESHOLD = 4;
String::String(const char* cp) {
    if (strlen(cp) <= STRING_THRESHOLD)
        rep = new Small_rep(cp);
    else
        rep = new Big_rep(cp);
}
```

其他的 String 成员函数只需要调用 String_rep 的成员函数就可以了，它们并不需要了解 String 到底是使用了 Small_rep 呢，还是使用了 Big_rep？

```
char String::operator[](int offset) const {
    return rep->operator[](offset);
}
```

即使是复制构造函数，它也不需要了解它所复制的是什么：

```
String::String(const String& s)
: rep(s.rep->copy()) {
}
```

这种通过指向基类指针来操作对象，而不去关心对象的具体类型的操作方式，正是面向对象编程的一个特性。我们在 String_rep 中定义了一系列的抽象操作集合，并在 String 的（除某个构造函数外）所有成员中使用了这些抽象操作。这使得 String 类更容易被维护及扩展；例如，如果我们希望增加一个新的 String_rep 实现，我们所需要在 String 中更改的惟一一个成员就是 String::String(const char*)；而且，String 的用户对这种改变根本一无所知。

3.5.2 会改变值的操作

如果类的值改变必须要求实现的改变，那么使用多种实现的方法就将变得十分棘手。例如，在我们的 String 类中，假设 Small_rep 要求字符串的长度不会超过 4 个字符，再假设我们的 String 类中还有一个成员函数是用来往已有的字符串里面添加字符的。那么当我们往一个 4 字符长的字符串中增添一个字符时，会有什么情况发生呢？

我们的方法是确保所有的 rep 都能够在所有可能的情况下正常工作——它们只是在性能特性上有所区别而已。往一个在创建时就有 4 个字符的字符串中添加一个字符的操作会稍慢于直接创建一个 5 字符长的字符串，但程序的运行结果仍然是正确的。

如果我们做不到这点的话，另外一个可供选择的方法就是：在每个会导致值发生改变的操作中创建一个新的 rep。这个新的 rep 是基于对象的新值来创建的：

```
String::String_rep*
String::build_rep(const char* cp) {
    if (strlen(cp) <= STRING_THRESHOLD)
        return new Small_rep(cp);
    else
        return new Big_rep(cp);
}

String::String(const char* cp)
: rep(build_rep(cp)) {
}

void
String::append(char ch) {
    /* 向已有的字符串末尾添加一个字符。
       首先，用这个新值来创建一个字符缓冲区 */
```

```
    const int len = length();
    char* newval = new char[len+2];
    for (int i=0; i<len; ++i)
        newval[i] = rep->operator[](i);
    newval[len] = ch;
    newval[len+1] = '\0';

    // 现在，删除掉原有的 rep 并创建新的 rep
    delete rep;
    rep = build_rep(newval);
    delete [] newval;
}
```

在 append 函数中，对于数据的复制是逐字符进行的——并且对于每个字符的获取都是通过一个虚拟函数来实现的。毫无疑问这种做法十分低效，我们应该如何加速它呢？我们可能会试图在 String_rep 中增加一个函数，它会返回一个指向字符串开始处的指针，但这将导致 String_rep 的所有实现中都必须将数据存储在一个连续的内存空间中。我们可能希望直接的实现能够避免这种限制，因此一个更好的办法就是定义一个函数 dump，它会将 String 中的字符都复制到一个调用方提供的字符缓冲区中：

```
class String_rep {
public:
// ...
    virtual void dump(char*) = 0;
};
```

这种方法不需要我们假定底层 rep 的实现细节；我们可以在每个不同的 rep 类中用最高效的方式来实现 dump。

在 rep 类中实现了 dump 后，我们就可以将 append 改写为：

```
void String::append(char ch) {
    /* 向已有的字符串末尾添加一个字符
       首先，用这个新值来创建一个字符缓冲区 */

    const int len = length();
    char* newval = new char[len+2];
    rep->dump(newval);
    newval[len] = ch;
    newval[len+1] = '\0';

    // 现在，删除掉原有的 rep 并创建新的 rep
    delete rep;
    rep = build_rep(newval);
    delete [] newval;
}
```

如同我们所见到的那样，使用多种实现会导致类变得更复杂；并且，由于每个操作都涉及到一次虚函数的调用，它还会带来运行时的开销。然而，多种实现可以允许类的设计者针对不同的实现进行优化，并对用户隐藏这种优化。

3.6 作为对象的句柄

在本章的所有例子中，我们都是使用了C++中的指针来作为我们的句柄。C++还允许我们实现功能类似于指针的类，并用这种“智能指针”来指向我们的rep。然而，要想很好地实现智能指针，我们还需要使用模板。因此，我们将在介绍模板（参见第7章）之后再来讨论该主题。

3.7 综述

句柄是C++中的一种普遍使用的实现技巧。它们赋予了类设计者足够的弹性来处理不定大小的实现，减少编译期的依赖，阻止他人获得类的实现细节，以及为同一个抽象数据类型提供不同的实现。所有的这些方法都会导致一些运行时的开销（至少是对每个操作有一个额外的内存引用），但通常情况下，与获得的好处相比，这种开销都是值得的。

3.8 小结

- 使用句柄来用常量大小的对象表示不定大小的值。
- rep类要么是一个嵌套类，要么是一个所有成员都为私用、但“真正”的类是它的友类的类。
- 使用计数器可以避免额外的复制操作，但我们还是需要对性能进行检测以确信使用它真的可以带来性能上的提高。
- 句柄可以用来确保对实现（而不是接口）的改变只会要求用户进行重新连接而不是重新编译。它们同样也可以对他人隐藏实现的细节。
- 句柄可以允许构造函数基于参数从一系列的实现中挑选合适的实现。
- 句柄不一定是一个普通的指针，它们也可以是“智能指针”对象。

3.9 问题

1. 请解释String的构造函数中对于缺省参数的选择。为什么我们会需要一个缺省值呢？

2. 如果我们往3.1节中的String类的构造函数传递一个空指针，会有什么样的情况产生呢？请给出一种用来处理该情况的更好办法，并重新实现String类来支持这种方法。

3. 重新实现我们的 String 类，把字符串的长度存储在 rep 中，而不是用一个 0 字节来作为字符串的结束标志。这样的改变会不会给我们的接口带来影响呢？（提示：是否存在用新的 String 类表示的字符串不能用原有的 String 类来表示？）这样的改变会给运行时的性能带来什么样的影响呢？

4. 在我们那个使用计数器的 String 类中，赋值操作符不得不对 s = s 这种情况进行检测；为什么我们不需要在复制构造函数中也对该情况进行检测呢？

5. 假设我们对那个使用了计数器的 String 类进行了修改，现在 String_rep 在创建时计数器的值是 1 而不是 0 了。这样的改动会给程序的性能和维护性带来什么样的影响呢？

6. 为什么我们不把 String::to_lower 改名为 String::tolower 呢？

7. 重新实现 3.2.2 小节的 String 赋值操作符，使得它里面不再出现判断和循环；并对前后两个版本的操作符进行性能和维护性方面的比较。

8. 在哪种情况下 String::to_lower（参见 3.2.3 小节）中的断言会失败？

9. 假设，在使用计数器的 String 类中，我们希望有一个成员函数：

```
class String {
public:
// ...
    char& operator[](int);
};
```

来允许我们这么做：

```
#include <ctype.h>
#include <String.h>
main() {
    String s("hello world");
    s[0] = toupper(s[0]);   // s == "Hello world"
    // ...
```

请在使用计数器的 String 类中实现该函数。

10. 对于 String 的多种实现版本，设计和实现一个 rep 类，它只被用来处理空字符串，不过它的效率非常高。（假设除赋值之外的所有 String 操作都不会改变该 String 的长度。）

11. String_rep::dump 会将字符存储到一个用户指定的字符缓冲区中。另外一种做法会在函数中为字符分配内存空间并返回指向该空间的一个指针。请对这两种做法进行比较。

第4章

继　承

许多对于继承的讨论都始于对语言规则的解释。当我们需要通过了解规则来使用这种特性时，我们应该首先确信我们对在设计的何处使用继承有着充分的了解。有着不恰当继承的程序也可以通过编译，但它们总是非常难以被理解和维护。

4.1　is-a 关系

继承应该用于“新类（派生类）描述的对象集是基类描述的对象集的一个子集”这种情况。这种关系就是 is-a 关系：

```
class Vehicle {
public:
    virtual double accelerate(double);
};

class Aircraft : public Vehicle {
public:
    virtual double accelerate(double);
};
```

所有 Aircraft 对象也都是 Vehicle 对象。（注意：反过来这个关系就不成立了；有些 Vehicle 并不是 Aircraft。）所有可以应用在 Vehicle 上的操作应用在 Aircraft 上应该同样有意义。（也就是说，基类的成员函数也可以由派生类来调用。）派生类可以通过重写的方式来在派生类中改变成员函数的实现，但它不应该改变该操作的概念模型。每个交通工具都可以被加速：自行车和火车在该操作上的实现方式虽然不一样，但在概念上来说，它们并无区别。

4.1.1　继承与组合

当基类是派生类描述的对象的一部分时，我们就不应该再使用继承了：

```
class Wing {
public:
```

```
    double length() const;
};

class Airplane : public Wing { // 不好的继承
// 此处忽略细节
};
```

Airplane 并不是一种特殊的、具有机身的 Wing；相反，它是由其他对象（包括 Wing）所组合而成的一种对象。（Airplane 不是 Wing，它有一个 Wing。）

这种对继承的不恰当使用使用户可以将 Wing 中的操作应用到 Airplane 上去：

```
Airplane a;
double l = a.length();
```

对 a.length()的调用会返回 Wing 的长度，而不是 Airplane 的长度。这并不是说上面的代码不能工作；但它会给我们带来迷惑：为什么我们要将机翼和发动机或者螺旋桨推进器区别对待呢？我们怎样才可以得到一架双翼飞机（它有着两对机翼）呢？

当对象是由其他对象组合而成时，正确的方法是将这些对象作为该对象的成员，而不是基类：

```
class Airplane {
    Wing w;
    Propeller p;
// 等等……
};
```

这意味着在 Airplane 中的 Wing 部分上的操作将会被明确地通过成员“w”进行调用；这也使得我们可以更清楚地阐述 Airplane 和 Wing 之间的关系：它们之间是 has-a，而不是 is-a 的关系。

4.1.2 在派生类中删除某些操作

所有在更通用的基类中的操作应该也能够应用到派生类的对象上去。虽然派生类可以通过重写基类的成员函数来为操作通过新的实现，它也不能通过将操作定义为私用的来删除在基类中合法的操作。

例如，如下（有着缺陷）的继承架构中为 Vehicle 提供了两种不同的 speed：普通的 speed()被用于地面交通；airspeed()用于空中交通：

```
class Vehicle { // 不好的设计
public:
    double speed() const;
    double airspeed() const;
};
```

```
class Land_vehicle : public Vehicle {
private:
    // 地面交通工具没有airspeed:
    double airspeed() const;
public:
    // 此处忽略细节
};
```

将 Land_vehicle::airspeed 重新声明为私用，意味着我们试图阻止外界对 Land_vehicle::airspeed 的调用：

```
class Car : public Land_vehicle {
// 此处忽略细节
};

Land_vehicle* vp = new Car;
double air = vp->airspeed();    // 编译期错误：airspeed 是私用的成员函数
```

但是这种尝试却失败了，因为我们还是可以通过 Vehicle*来调用这个被我们禁止调用的成员函数：

```
Vehicle* vp = new Car;
double air = vp->airspeed();    // 合法的操作
```

任何试图在派生类中对操作进行限制的行为通常都意味着继承架构设计上的失误。为了从我们的例子中将这个失误挑选出来，我们需要对“将 airspeed()应用到陆地交通工具上是否有意义”这个问题做出判断。如果我们认为它没有意义，那么我们就不应该把 airspeed()声明在 Land_vehicle 的任何基类中去。相反的是，我们应该把它移到继承架构中的某处，对从那开始的类及其派生类生成的对象来说，airspeed 是一个有意义的操作。例如，我们可以创建一个新类 Air_vehicle，所有有着空中速度的交通工具都由它派生而来：

```
class Vehicle {
public:
    double speed() const;
};

class Land_vehicle : public Vehicle { /*...*/ };

class Air_vehicle : public Vehicle {
public:
    double airspeed() const;
    // 此处忽略细节
};
```

通过使用该架构，我们就不能再对 Land_vehicle 调用 airspeed 了，即便是通过 Vehicle*也不行。(在本章结束处，问题 1 中给出了另外一种用来修正该架构的方法。)

4.1.3 私有、保护以及公有继承

一个类可以向外给出两个接口：一个面向用户（类的公用部分）；另外一个则面向派生类的实现（类的公用部分以及保护部分）。继承也同样如此：如果继承方式是公有的，它就是面向对象的接口的一部分，用户也就可以以此编写依赖于该继承关系的代码。如果继承方式是保护型的，那么它只是面向派生类实现的接口的一部分；如果继承方式是私有的，它将不属于任何的接口——只有类的实现本身（以及其友元）才可以使用这种继承关系。

回顾：公有、保护以及私有继承

在 C++中有着三种不同的继承方式：公有、保护和私有。在所有的继承方式中，派生类的成员都可以访问基类中的公用和保护型成员——但不能访问它的私用成员。这三种不同的继承方式之间的区别在于派生类向其用户提供的接口（相对于派生类的实现来说），以及用户何时才可以隐式地将一个指向派生类的指针转化为一个指向基类的指针。

最常见的继承方式就是公有继承：

```
class Shpae {
public:
    Color color();
};

class Circle : public Shape {
// 此处忽略细节
public:
    Circle(unsigned);
    unsigned radius() const;
};
```

当使用了公有继承后，基类中的公用成员也就同时成为了派生类中的公用成员，它里面的保护型成员也成为了派生类中的保护型成员：

```
Circle c(5);     // 构造一个半径为 5 的圆
c.color();       // OK - color()是公有基类中一个公用成员
```

一个指向派生类的指针可以被隐式地转化为一个指向公有基类的指针：

```
Shape *sp = new Circle(5); // OK - Shape 是一个公有的基类
```

当私有继承被使用时，基类中的公用成员以及保护型成员都变成派生类中的私用成员。我们可以通过派生类中的成员和友元来访问它们，但是用户就不行了：

```
class String {
public:
    String(const char* = "");
```

```
    char operator[](int i) const;  // 第i个字符
    int length() const;
};

class Telephone_number : private String {
public:
    Telephone_number(const char*);
    int uses_special_keys() const;
};
```

Telephone_number 中的成员可以使用 String 中的公用和保护型成员：

```
#include <ctype.h> // for isdigit()

int Telephone_number::uses_special_keys() const {
    for (int i = 0; i < length(); ++i)
        if (!isdigit((*this)[i]))
            return 1;
    return 0;
}
```

然而，Telephone_number 的用户就不能调用 String 中的任何成员函数：

```
main() {
    Telephone_number tn("5551212");
    int len = tn.length(); // 编译期错误：length 是私用成员
```

用户不能隐式地将一个指向派生类的指针转化为指向私有基类的指针：

```
void f() {
    Telephone_number s("5551212");
    String* ptr = &s;   // 编译期错误：String 是一个私有基类
}
```

如果我们使用的是保护继承，基类中的保护型和公用成员都将成为派生类中的保护型成员。派生类可以调用保护型基类中的成员函数，并且也可以隐式地将一个指向派生类的指针转化为一个指向保护型基类的指针。

通过使用存取声明指令，私有或保护型基类中的公用对象也可以变成派生类中的公用成员：

```
class String {
public:
// ...
    int length() const;
};

class Telephone_number : private String {
public:
```

```
// ...
    String::length; // Access declaration
    String::length[1];    // 存取声明指令
};
```

现在用户就可以调用 Telephone_number 的成员函数 length 了（就好比我们进行了如下的声明）：

```
class Telephone_number : private String {
public:
// ...
    int length() { return String::length(); }
};
```

4.2 公有继承

当继承是接口的一部分时，我们就应该使用公有继承。也就是说，我们希望向用户讲解 X 和 Y（X 派生自 Y）之间的 is-a 关系。和接口的其他部分一样，我们（在某种程度上来说）也保证了永远不会对类中的这部分进行修改！这是因为用户可能会编写依赖于“从指向 Derived 的指针到指向 Base 的指针之间的隐式转换”的代码：

```
void set_color(Shape&, Color);
Circle c(1);          // 1cm 圆周
set_color(c, blue);
```

上面的代码依赖于“Circle 是一种 Shape，因此指向 c 的应用可以被当作 Shape&传递给其他接受 Shape&参数的函数”这个事实。这意味着我们不可能在随后的改动中从该类中删除掉这个继承关系并同时保证已有代码可以正常工作！这样的改动对于接口来说是不允许的——如我们无法移除一个公用的成员函数。

4.3 私有继承

当继承不是接口的部分（而是实现的细节）时，我们应该使用私有继承。用户不可能写出依赖于这种继承方式的代码，这使得我们可以在不需要继续使用已有的基类时对实现进行修改。

由于组合（将“基类”声明为一个数据成员）和其非常相似并且在通常情况下工作得也很好，所以相对公有继承来说，私有继承的私有要少得多。与从一个基类上继承相比，我们可以将一个基类的对象声明为（原）派生类的成员。这样做不会带来任何运行时间和空间上的开销，并且得到的类也容易让人理解（阅读代码的人们不一定会去记忆哪个成员函数是从它的私有基

1 译注：现在通用的做法是使用 using 指令来实现将原本不是公用的基类中的成员在派生类中成为公用的。

类中继承而来的这个问题)。

例如，在前面的复习中的 Telephone_number 类就应该被重新实现为：

```
#include <ctype.h> // for isdigit()
class String { // 同前——没有任何改动
public:
    String(const char* = "");
    char operator[](int i) const;  // 第 i 个字符
    int length() const;
};

class Telephone_number {
private:
    String s;
public:
    Telephone_number(const char*);
    int uses_special_keys() const;
};

int Telephone_number::uses_special_keys() const {
    for (int i = 0; i < s.length(); ++i)
        if (!isdigit(s[i]))
            return 1;
    return 0;
}
```

Telephone 成员函数代码中惟一的改变就是：以前可以隐式地使用基类 String 中的成员现在必须要通过限定格式的形式来使用了。这种改变并不会影响到我们的用户：他们代码的运行结果并不会因此而有所不同；并且程序的运行效率和占用内存空间的大小也不会改变。不论我们用哪种方法，Telephone_number 的对象中必须包含一个 String 的对象。

在大多数情况下，没有基类的类总是要比和它功能相同但却使用了私有继承的类要容易理解和扩展。如果我们使用了组合这种方式，那么当以后我们希望给这个类增加一个新的基类时，我们只会得到一个简单的继承架构，而不会得到复杂的多重继承架构。在很多的架构中，使用多重继承的代码总是比使用简单继承的代码要慢，占用的空间也要大一些，并且还会使我们对它的理解过程变得更难。

在如下情况下，我们将会得到该规则的一个特例：我们在派生类中需要重写基类中的虚函数，但又不希望对基类的使用暴露于公用接口中。如果被重写的函数是析构函数时，最简单（也是惟一）的做法就是使用私有继承。

例如：假设我们使用的 C++运行环境支持对派生自 Collectible 的对象进行自动的垃圾收集。当我们调用 garbage_collect()时，那些无法使用已有指针来访问的可收集对象就将会被删除：

```
class Collectible {
public:
    Collectible();
    virtual ~Collectible() {}
};

void garbage_collect() {
    Collectible* cp;
    while (cp = find_unreachable_object())
        delete cp;
}
```

再假设我们设计了一个垃圾收集类来表示一幅图中的节点。虽然我们可能难以对我们的用户隐藏“节点是通过垃圾收集机制被删除”这个事实，我们仍然可以选择我们的垃圾收集器。为了做到这一点，我们把 Collectible 声明为一个私有的基类：

```
class Node : private Collectible {
public:
    virtual ~Node();
    // 此处忽略细节
};
```

通过重写虚拟的析构函数，我们可以确保在 garbage_collect 中 delete cp;这条语句会调用被收集的 Node 的析构函数。通过使用私有继承，我们还可以保留改变 Node 实现的选项来使用其他的垃圾收集机制。

4.4 保护型继承

当继承是面向派生类而不是面向用户的接口中的一部分时，我们就选择使用保护型继承。一个保护型的基类和一个能够被所有派生类所访问的私有基类差不多：

```
class String { /* ... */ };

class Telephone_number : protected String { /* ... */ };

class Local_number : public Telephone_number { /* ... */ };
```

在类 Local_number 中的成员函数都可以访问它们的 String 部分的公用和保护成员。

我从来都没有使用过保护型继承，也没有听说有人在哪个项目中使用过它。除了会得到一个保护型基类之外，那些用来阻止使用私有继承的原因同样也适用于保护型继承，而且我们也可以很容易就得到一个保护型成员：

```
class String { /* ... */ };
```

```
class Telephone_number {
protected:
    String s;
    // 此处忽略细节
};

class Local_number : public Telephone_number { /* ... */ };
```

上面的代码大大地简化了继承架构。并且它还没有改变代码的运行效率，Local_number 中的成员函数仍然可以访问到对象中的 String 部分（此时惟一的改动就是我们必须通过 s 来访问它们）。

我们并不是说保护型继承没有用——如果派生类中的成员需要重写（保护型）基类中的虚函数，我们就可以使用保护型继承了。但如果可以使用组合，我们就应该使用组合：因为使用语言中的晦涩特性（如保护型继承）会增大理解程序的难度。

4.5 与基类抽象的一致性

通过使用相同的名字和参数来进行重新声明，派生类可以重写基类中的虚函数：

```
class Vehicle {
public:
    virtual void accelerate(double);
    double speed();
};

class Car : public Vehicle {
public:
    virtual void accelerate(double);
};

class Submarine : public Vehicle {
public:
    virtual void accelerate(double);
};
```

然而当继承被使用时，对于 Car 和 Submarine 类中的成员 accelerate，除了要让所有类型都正确外，我们还有着一个严格的限制：*派生类中的成员函数必须和基类的抽象模型相一致*。虽然实现可能会不一样，但所有派生自 Vehicle 的对象都应该“像 Vehicle 那样加速”。这种语义上的限制并没有出现在 C++的语法规则中，也没有 C++编译器能够检测出它们来。

对于 Vehicle 来说，关于 accelerate 的抽象模型必须指出：对于交通工具的加速会导致它速度上的改变，如：

$$accelerate(x) \Longrightarrow (speed_{new} == speed_{old} + x)$$

一旦我们将这种抽象记录到文档中后，所有的派生类的行为都应该遵循该抽象模型。

为什么要这样做呢？如果所有的派生类都遵循该抽象模型，用户就可以写出如下的依赖于抽象模型的代码：

```
void
full_stop(Vehicle& v) {
    v.accelerate(-v.speed());
}
```

并将这些代码应用到所有的 Vehicle 之上：

```
Submarine trident;
full_stop(trident);

Car volkswagen;
full_stop(volkswagen);
```

当新的 Vehicle 派生类被增加时，我们可以将它们应用到这些已有的代码中去，并且也能够得到正确的结果！

```
class Aircraft : public Vehicle {
public:
    virtual void accelerate(double);
};

// ...
    Aircraft boeing_747;
    full_stop(boeing_747);  //希望我们还在!
```

通过对同一类抽象操作（虚函数）的调用，我们就可以用一个函数来处理所有继承自 Vehicle 的派生类对象，并在函数不了解这些对象的实现细节的情况下得到正确的结果。这也是面向对象设计给我们带来的一个主要好处。

如果 speed 和 accelerate 之间的关系并没有被记录到文档中去，并且开发人员也不清楚这种关系，那么我们就有可能得到一个不遵循该抽象模型的派生类。例如：

```
class Hot_rod : public Car {
public:
    virtual void accelerate(double);
};

void
Hot_rod::accelerate(double delta) {
    // Hot_rod 的加速度非常大!
    Car::accelerate(2*delta);
}
```

上面代码中的 Hot_rod 设计者就误解了成员函数 accelerate 的含义，因此设计出这样一个不遵循 Vehicle 的抽象模型的 Hot_rod 类。

当我们对一辆速度为 100 公里/小时的 Hot_rod 调用 full_stop 时，会有什么现象产生呢？函数 full_stop 会调用：

```
v.accelerate(-v.speed());
```

在本例中，它会去调用：

```
Hot_rod::accelerate(-100);
```

接下来就是调用：

```
Car::accelerate(-200);
```

当我们调用了 full_stop 后，Hot_rod 就将以 100 公里/小时的速度后退！这并不是我们所期望的。虽然上面的代码完全符合语言规则（它可以被正确地编译），但它仍然是错误的：因为 Hot_rod 并没有遵循 Vehicle 的抽象模型。

4.6 纯虚函数

我们 Vehicle 类的最初声明中包含了一个对成员函数 accelerate 的声明。我们之所以将它声明在 Vehicle 中是因为，在概念上来说该操作可以适用于所有的交通工具。我们希望在随后的每个派生类中，该函数都会被重写。

那么我们该如何实现 Vehicle::acclerate 呢？我们不希望出现 Vehicle 对象。我们只是将 Vehicle 作为基类来描述一系列在派生类中普遍存在的概念。我们希望 Vehicle 只能被当作基类使用，并且所有的派生类都会重写 accelerate。除此之外，我们甚至不希望看到有人调用 Vehicle::accelerate。一种可行的办法就是：我们在每次的 Vehicle::accelerate 中都输出一条错误信息来禁止别人对它的调用：

```
void
Vehicle::accelerate(double) {
    cerr << "Vehicle::accelerate called?\n";
    abort();
}
```

但这种方法只能检测到运行时重写 accelerate 的错误。在 C++中，还有一种特性可以更好地实现我们的要求，它可以在编译时期就发现这种类型的错误——它就是纯虚函数。

回顾：纯虚函数和抽象基类

在虚函数声明的参数列表后面加上“=0”：

```
class C {
    virtual void f() = 0;
};
```

就将使该函数变成纯虚函数。我们在此不需要为纯虚函数 C::f()提供任何的定义。那些声明（或继承）了纯虚函数的类就是抽象基类。任何试图创建一个抽象基类对象的操作都将导致编译期错误的产生。

如果一个类派生自 C 并重写了 C::f，它就将成为具体（非抽象）类：

```
class D : public C {
    void f();
};
```

我们通常都将抽象基类用于接口的声明，这时我们不需要为该接口声明一个完整的实现。该接口描述了所有派生自该类的对象都应该支持的抽象操作；派生类必须实现这些抽象操作。例如：

```
class Vehicle {
public:
    virtual double accelerate(double) = 0;
    virtual double speed() = 0;
};
```

由于 Vehicle 是一个抽象类，任何创建 Vehicle 对象的尝试都会导致编译期错误的产生：

```
Vehicle v;  // 编译期错误："Vehicle" 是抽象类
```

为了使用 Vehicle，我们必须从它派生出其他的具体类：

```
class Car : public Vehicle {
public:
    virtual double accelerate(double);
    virtual double speed();
};

class Bicycle : public Vehicle {
public:
    virtual double accelerate(double);
    virtual double speed();
};
```

由于在 Car 和 Bicycle 中，基类中的所有纯虚函数都被重写了，所以我们就可以得到由这两个类创建出来的对象。

即使我们得不到 Vehicle 对象，我们仍然可以使用指向 Vehicle 的指针和引用：

```
void
full_stop(Vehicle& v) {
```

```
    v.accelerate(-v.speed());
}
```

一个继承（而没有重写）了纯虚函数的类同样也是一个抽象类：

```
class Land_vehicle : public Vehicle {
};

Land_vehicle v; // 编译期错误：Land_vehicle 是一个抽象类
```

通过将 Vehicle 声明为抽象类，编译器就会对那些试图创建 Vehicle 对象的操作产生出一个编译期错误。我们并不需要为基类中的成员函数编写那些讨厌的存根函数（stub function）。基于这些原因，对那些类似于 Vehicle 这种“用来描述一系列的派生类”的类，我们总是推荐使用纯虚函数和抽象类的做法。

析构函数不能声明为纯虚的：

```
class Vehicle {
public:
    virtual ~Vehicle() = 0; // 不好的做法
    // 此处忽略细节
};
```

在 Vehicle 的任意派生类中，在调用它们的析构函数时都会调用到 Vehicle::~Vehicle。由于我们必须为该析构函数提供一个定义（否则我们就会得到一个链接错误），因此将它声明为纯虚函数显得毫无意义。

4.7 有关继承的细节和陷阱

C++中对于继承的处理还有着一些“小窍门”，我们现在就来讨论它们。

4.7.1 没有被继承的东西

当我们使用继承时，我们必须时刻牢记，下面的这些东西没有从基类中被继承：

- 构造函数（包括复制构造函数）。如果我们没有声明复制构造函数，那么编译器将会自动为我们创建一个。在这个被创建的复制构造函数中，它会调用类中所有非静态的数据成员以及基类的复制构造函数[2]。
- 析构函数。如果我们没有声明析构函数，并且所有的非静态数据成员和基类都有析构函数，编译器就会为我们自动生成一个这样的析构函数：它会调用那些非静态数据成员以及基类中的析构函数。如果类中的某个基类的析构函数是虚函数，那么这个合成

2 译注：有关编译器合成的构造函数、复制构造函数、析构函数等成员中的调用顺序，请参见 Stan Lippman 所著的《Inside C++ Object Model》。

的析构函数也将是虚函数。

- 赋值操作符。如果我们没有声明赋值操作符，编译器也会为我们自动生成一个：它会去调用所有非静态数据成员和基类中的赋值操作符。
- 被隐藏的成员函数。如果在基类中存在的成员函数在派生类中没有被重写，并且在派生类中还声明了一个和该函数有着相同名字但参数列表不同的成员函数，那么在基类中的那个成员函数就将被隐藏。例如：

```
class Car {
public:
    // 忽略其他内容
    void steer(int degrees);
};

class Autopilot {
public:
    Autopilot();
};

class Smart_car : public Car {
public:
    // 忽略其他内容
    void steer(Autopilot&); // Car::Steer(int)被隐藏了
};
```

我们的 Smart_car 中可以通过自动驾驶仪来进行驾驶，但我们却把基类中的 Steer 给隐藏掉了：

```
Smart_car c;
c.steer(45);    // 编译期错误：无法将 Autopilot 转化为 int
```

如果我们不希望隐藏基类中的函数，我们就应该在派生类中对它们进行重新声明：

```
class Smart_car : public Car {
public:
    // 忽略其他内容
    void steer(int i) { Car::steer(i); }
    void steer(Autopilot&);
};
```

在某些 C++编译器中，如果派生类会隐藏基类中的函数，它就会向外产生一个警告信息。

4.7.2 在派生类中指定 virtual 关键字

当我们重写一个虚函数（不管它是不是纯虚函数）时，我们不需要明确地使用 virtual 这个关键字；编译器会自动将这个和基类中的虚函数有着同样的名字及参数列表的函数视为虚函数：

```
class Base {
public:
    virtual void f();
};

class Derived : public Base {
public:
    void f();   // 等同于"virtual void f()"
};
```

不过我们最好还是指定 virtual 这个关键字，这会使那些阅读我们代码的人更容易了解我们的代码。并且使用 virtual 与否都不会对程序的意思带来任何影响。

4.7.3 在构造函数和析构函数中调用虚函数

当我们在构造函数和析构函数中调用虚函数时，它的运行结果可能会有所不同。当构造函数正在创建派生类中的基类部分时，被构造的对象就被视为基类（而不是派生类）的一个对象。这意味着我们调用的虚函数将会是正在被构造的基类中的那个成员函数，而不是派生类中的成员函数。

例如，

```
class Base {
public:
    Base();
      virtual void debug_print()
        { cout << "Base::Base();\n"; }
};

class Derived : public Base {
public:
    Derived();
      virtual void debug_print()
        { cout << "Derived::Derived();\n"; }
};

Base::Base() {
    debug_print();
}

main() {
    Base b;
    Derived d;
}
```

上面的程序会输出：

```
Base::Base();
Base::Base();
```

即使我们正在构造的是 Derived 中的 Base 部分，在 Base 的构造函数中调用的 debug_print 也会是 Base::debug_print()。

出现这种令人迷惑的行为的原因是：对象的基类部分的构造要早于其数据成员。当 Derived 中的 Base 部分被构造时，Derived 中的其他数据成员还都没有被构造。此时调用 Derived 中的虚函数将变得毫无意义：因为它可能会去试图访问 Derived 中那些还没有被初始化的数据成员。（换个角度来看，当 Base 的构造函数被调用时，正在被构造的对象实际上不是 Derived，因此我们也就不可能调用到 Derived 的成员函数。）

该逻辑也适用于在析构函数中调用虚函数这种情况：

```
Base::~Base() {
    debug_print();
}
```

在上面的析构函数中，我们调用的总是 Base::debug_print，即使当被摧毁的是 Derived 中的 Base 部分时也是如此。当我们调用 Base 的析构函数时，Derived 中的数据成员已经早就被析构了，因此再去调用 Derived 中的 debug_print()也将变得毫无意义。

记住：只有在构造和析构的过程中调用虚函数才会导致这种特殊的行为出现。在其他情况下（即使是我们在构造函数或析构函数中调用它时），虚函数的行为都是正常的：

```
Base::Base() {
    debug_print();  // 调用的是 Base::debug_print()
    Base *bp = new Derived;
    bp->debug_print();  // 调用的是 Derived::debug_print()
}
```

4.8 小结

- 继承描述的是 is-a 关系：派生类所实现的对象应该是基类所实现的对象集中的一个子集。
- 当继承是接口的一部分时，请使用公有继承。只有当继承被用来隐藏实现细节时，我们才需要使用私有和保护型继承。
- 对于私有继承的大部分使用，我们都可以用组合来替代；只有当派生类需要重写（私有）基类中的虚函数时，我们才不能那样做。
- 在派生类中被重写的虚函数应该和基类中的抽象模型相一致。
- 构造函数、析构函数以及赋值操作符不能被继承。

4.9 问题

1. 假设在我们的 Vehicle 继承架构中，我们希望将陆地交通工具的 airspeed 定义为它的 speed。这样做会对 Vehicle 架构带来什么样的影响？它是否会使类更容易被使用？

2. 我们应该如何改编我们的 Vehicle 类，以确保所有派生类中对 accelerate 的实现都会遵循它的抽象模型？（提示：我们应该让派生类重写一些其他的函数，而不是 accelerate。）我们这样做会给运行时的效率带来什么样的影响？

3. 我们可以（但不是必须）定义一个纯虚函数；对于这样的虚函数，我们只能通过使用:: 指定符来直接调用它们：

```
Car c;
c.Vehicle::accelerate(20);
```

我们应该如何来使用该特性？是否还存在着另外一种更好的办法来让我们避开语言中的晦涩特性？

第 5 章

多重继承

C++中支持多重继承：一个类可以有两个或更多基类。在本章中，我们将讨论那些使用多重继承的设计是如何来对对象间公有的 is-a 关系进行模塑的。虽然这并不是多重继承的惟一使用方法，但它却是最容易被程序员理解和使用的方法：因为相对用于模塑 is-a 关系的简单继承来说，这种扩展显得很自然。我们将侧重于理解为什么要使用多重继承来进行设计这方面：大部分的和语言相关的细节都被我们放到“复习”中去了。

作为例子，我们将会设计一系列的对象来表示 C++程序中的不同部分。例如，我们可能会用一种对象来表示类，一种用来描述指针类型，另外一种用来描述函数。这样 C++编译器或者其他工具就可以使用它们了。

5.1 作为交集的多重继承

在公有基类中封装的概念还会经常出现在某些派生类中；我们把这些类称为基类的派生集。举例来说，在编写编译器时，人们可能会用一个基类 Type 来封装那些用来表示存在于所有 C++类型中的公共概念。Type 的派生集包括 C++中各种不同的类型，如：内建类型、指针、以及引用等。

回顾：多重继承

一个类可以有多个基类；其中的每个基类都可以是公有、私有或者是保护型的基类。

```
class Derived : public Base1,
    protected Base2,
    private Base3 {
};
```

在上例中，Base1 是 Derived 的一个公有基类；Base1 中的公用和保护型成员也就变成了 Derived 中的公用或保护型成员，我们还可以隐式地从一个 Derived*通过转换得到一个 Base1*。Base2 是 Derived 的一个保护型基类；它里面的公用和保护型成员也就变成了 Derived 中的保护型成员。Base3 是 Derived 的一个私有基类；它里面的所有公用和保护型成员都成了 Derived

中的私用成员。

如果一个有着同样名字的成员存在于两个以上的基类中，并且派生类中没有定义该成员，那么对这个成员使用末限定形式的访问将会产生歧义：

```
class Telephone {
public:
    void reset();
};

class Computer {
public:
    void reset();
};

class Smart_telephone
: public Telephone, public Computer {
};

main() {
    Smart_telephone t;
    t.reset();  // 编译期错误：哪个 reset?
}
```

在上面的例子中，我们试图在 Smart_telephone 上面调用 reset，但编译器却告诉我们在解析 reset 时碰到了歧义；因为它的两个基类中都定义了自己的 reset()。通过在调用函数时为该函数指定基类名的方式，我们可以去除掉这种歧义：

```
Smart_telephone t;
t.Computer::reset();    // OK
```

另外一个更通用的用来消除歧义的方法就是在派生类定义一个成员函数，用它来完成我们想要做的事情：

```
class Smart_telephone
: public Telephone, public Computer {
public:
    void reset();
};

void Smart_telephone::reset() {
    Computer::reset();
    Telephone::reset();
};

main() {
    Smart_telephone t;
```

```
    t.reset();  // OK
}
```

现在，Smart_telephone::reset()就将基类中的 reset 给隐藏了，因此我们对 Smart_telephone 调用 reset 也就不会再得到歧义了。

对于这种事情，我们可以分两方面来看待它。图 5.1 中给出了一个典型的继承树，其中 Type 作为基类被放到树的顶端，下面的就是它的派生类。（在这幅图中，我们还省略了一些 C++中的类型以避免在图中造成混乱。）

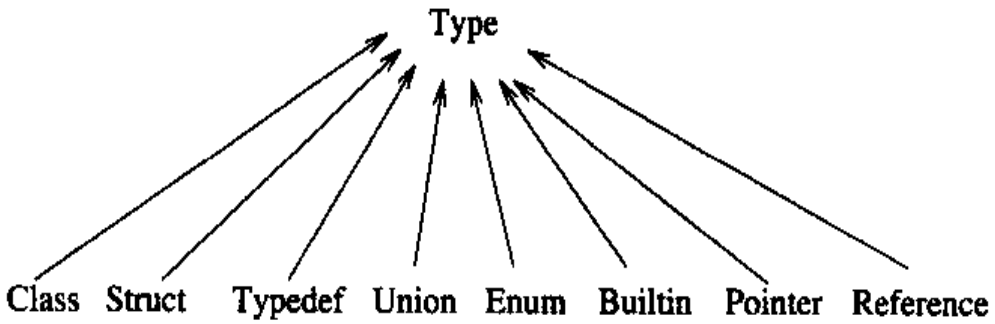


图 5.1 关于类型的继承树

图 5.2 中展示的是同一继承架构的另外一种表现形式。这种用来表示所有对象间关系的图也被称为维恩图，从中我们可以看出，由于 Class 是 Type 的一个子集，所以每个 Class 的对象同时也是一个 Type 的对象。在图中，区域的大小和形状对于它们所表示的对象来说无关紧要。

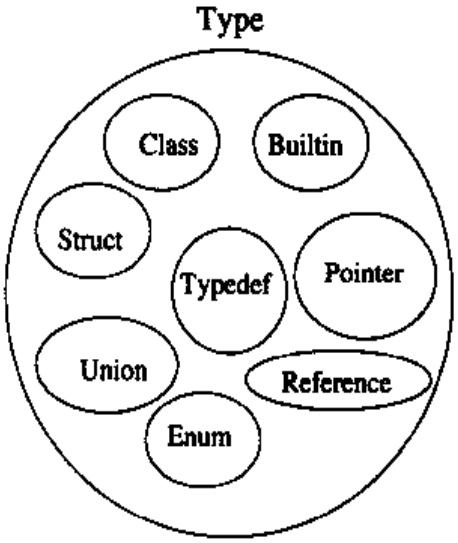


图 5.2 关于类型的维恩（Venn）图

假设我们在以后可能希望增添一个新的基类，用于在使用 C++中的对象被使用前的声明；这样的类可能会封装一下有关名字匹配的概念。我们希望所有的对象（包括函数、数据、假名、类、结构、联合、枚举等）都必须在使用前被声明。图 5.3 中展示的就是关于这样一个派生集合的维恩图。

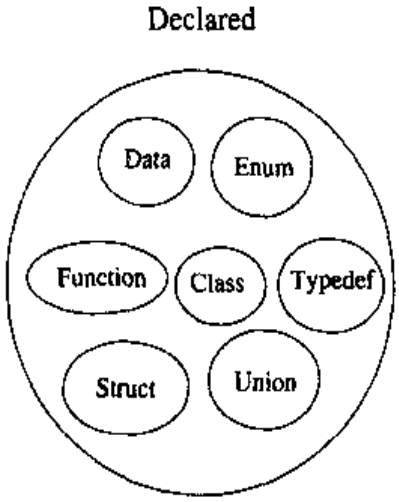


图 5.3 用于声明对象的维恩图

现在考虑一下，当我们试图同时在一个架构中使用 Type 和 Declared 时会发生什么呢？如同我们在图 5.4 中所看到的，从 Type 和 Declared 中分别派生出的集合之间有着一定的交集。

不管是 Type 还是 Declared，它们都不可能成为另外一方的基类：这是因为有些属于 Type 的对象（如内建类型、指针以及引用）并不属于 Declared；同时，在 Declared 中也有一些对象（如函数和数据）不属于 Type。然而，还是存在一些对象，它们既是 Type，同时也是 Declared，如：Class、Struct、Typedef、Union 以及 Enum。多重继承正好可以用来描述这种模型。作为 Type 的 Class 同时也是 Declared，因此我们就将这两个类作为它的基类：

```
class Class : public Type, public Declared {
// ... 细节被忽略
};
```

现在，Class 就能同时提供 Type 和 Declared 所能提供的所有操作了。

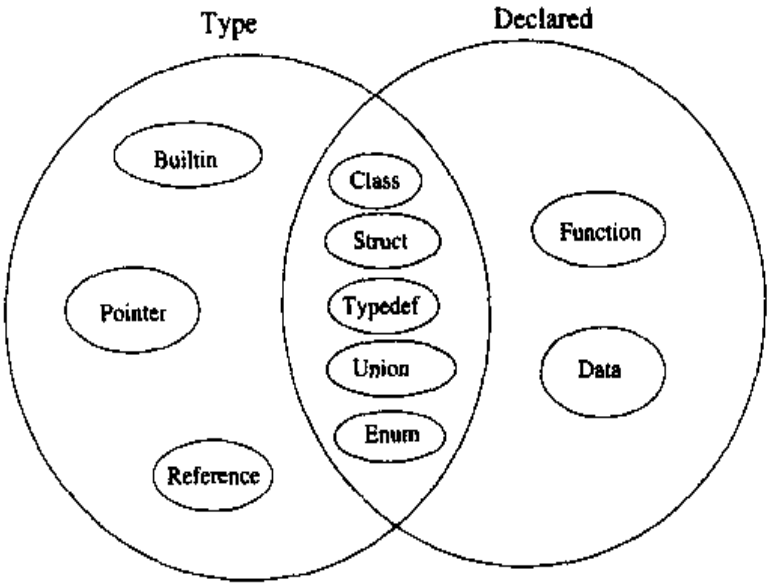


图 5.4 用于描述 Type 和 Declared 之间交集的维恩图

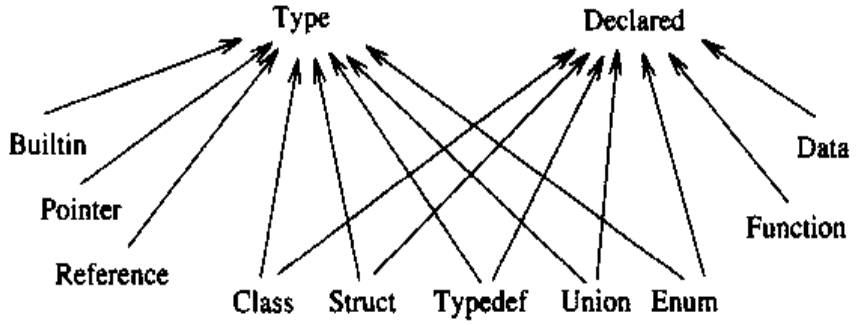


图 5.5 对于多重继承的第一次尝试

如果对所有被声明的类型进行上述操作，我们就可以得到图 5.5 中所描述的架构。虽然该架构可以很好地表达出 is-a 的关系，但它还是显得有点不实用。我们可以对它进行一定的简化——创建一个新类 Declared_type，用它来表示所有既是 Declared 又是 Type 的类：

```
class Declared_type : public Type, public Declared {
// 细节被忽略
};

class Class : public Declared_type {
// ...
};

class Struct : public Declared_type {
// ...
};

// 等等
```

这样我们就可以得到如图 5.6 中所展示的继承架构。

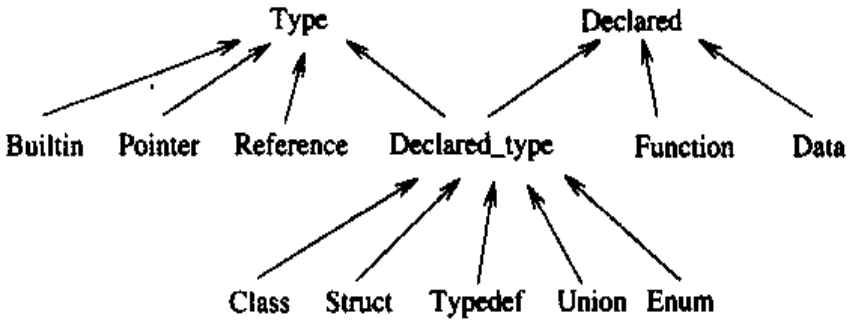


图 5.6 修改后的继承树

由于只有一个类有着多个直接的基类，所以图 5.6 中所示的架构更为简单，并且也更容易被维护。例如，如果在 Type 和 Declared 中有着同样名字的成员（这会导致歧义性的产生），我们只需在一个类（Declared_type）中进行修改就可以很好地处理这个问题。

5.1.1 什么时候不应该使用多重继承

多重继承描述的是对象间的交集，而并不是合集。当我们这样写时：

```
class D : public B1, public B2 {
```

我们就已经断定在 B1 所描述的对象和 B2 所描述的对象间存在着一个非空的交集，而 D 就是这个交集。我们可以在 D 上面调用所有 B1 和 B2 的公用成员函数；因此 D 不但是 B1，同时也是 B2。

如果 B1 和 B2 之间没有交集的话，我们就不能使用多重继承。特别地，并非 D 要么是 B1，要么是 B2，而恰好同时既是 B1，也是 B2。

5.2 虚基类

当使用多重继承来模塑 is-a 关系时，如果派生类中可能包含同一基类中的多份拷贝，我们就应该使用虚基类。在两个类中存在多份的 is-a 关系没有任何意义：X 和 Y 之间的 is-a 关系只能是存在或否。说两次“X 是 Y”毫无意义。

现在让我们来看看该如何修改我们的设计以导入虚基类。假设 C++程序中的所有部分都存在着同样的一种功能：我们希望用一个通用的基类（Program_part）来在我们的设计中体现这一特性。如图 5.8 所示，所有 Type 和 Declared 的对象同时也是 Program_part 的对象。

该如何制订我们的架构呢？我们可能会试图将 Program_part 作为 Type 和 Declared 的一个基类：

```
class Type : public Program_part {
// 此处忽略细节
};

class Declared : public Program_part {
// 此处忽略细节
};
```

但这样做并不能达到我们的期望：因为现在每个 Declared_type 对象中将同时具有两个 Program_part 子对象，所以我们必须将 Program_part 定义为一个虚基类：

```
class Type : public virtual Program_part {
// 此处忽略细节
};

class Declared : public virtual Program_part {
// 此处忽略细节
};
```

回顾：虚基类

考虑如下的代码：

```
class A {
public:
    int i;
};

class B : public A{};
class C : public A{};
class D : public B, public C{};
```

在每个B，或者C的对象中都有一个A子对象；并且在D的对象中同时存在着B和C的子对象，那么在每个D的对象中究竟有几个A子对象呢？

在上面的代码中，一个D对象中同时存在两个不同的A子对象。因此，对于D中的A子对象的引用就必须通过限定的形式来进行（否则就会引起歧义）：

```
main() {
    D d;
    d.i;        // 存在歧义
    d.A::i; // 存在歧义：到底是哪个A呢？
    d.B::i; // OK - B子对象中的A子对象的i
    d.C::i; // OK - C子对象中的A子对象的i
}
```

在很多的设计中，同时具有两个A子对象并不是一件正确的事情；相反，我们应该让B和C这两个基类同时共享一份A子对象。我们可以通过把A声明为一个虚基类来做到这一点：

```
class A{};

class B : public virtual A{};
class C : public virtual A{};
class D : public B, public C{};
```

对象中所有虚基类的实例都是被共享的；在本例中，B和C共享了同一份A子对象。如果在一个对象中同时包括了同一基类的虚拟和非虚拟版本，那么只有虚基类才会被共享。

虚基类通常都是由最外层派生类来构造的。在我们的例子中，用来创建A子对象的责任落到了D的构造函数上；在B和C的构造函数中，用于初始化基类A的那部分代码则不会被执行。

将指向虚基类的指针转化为指向任意派生类的指针这种做法是合法的：

```
main() {
    D d;
    A* ap = &d;     // OK
```

```
    D* dp = (D*)ap  // 编译期错误
}
```

虚基类中的名字解析

当架构中包含虚基类时，用来进行歧义解析的规则也就变得越复杂。当多个基类中都包括同样名字的成员，并且我们在派生类中使用该成员时，编译器会从多个名字中挑选出最适合的那个名字。当B是D的（直接或间接）基类时，D::x在解析过程中就要优于B::x。例如：

```
class Base {
public:
    int x;
};

class D1 : public virtual Base {
public:
    double y;
    short z;
};

class D2 : public virtual Base {
public:
    long x;
    void* y;
    char* z;
};

class D12 : public D1, public D2 {
public:
    float z;
};
```

在上面的例子中，D12::z要优于D1::z，但D1::y和D2::y之间不存在着优先级关系。实际上，用来描述这种优先级关系的最好做法就是画图（见图5.7）。由于从D12::z出发我们可以找到所有的z，所以D12::z也就优于其他的z；但对于成员y来说就不存在这样的关系了，因为我们找不到从一个y到另外那个y之间的路径。

如果没有一个名字优先于其他的名字，我们就会得到一个编译期错误：

```
D12 d12;

d12.x;  // D2::x，它要优先于Base::x

d12.y;  // 编译期错误：D1::y和D2::y之间没有优先级关系

d12.z;  // D12::z，它优先于D1::z和D2::z
```

```
d12.D1::y;  // 明确地指定这种做法是可行的
```

用来判断名字间优先级关系的规则不会对存取（私用、保护型或者公用）规则有任何的影响。即便是在代码中存在歧义的情况下，我们也可以对涉及到的那一个名字进行正当的存取。

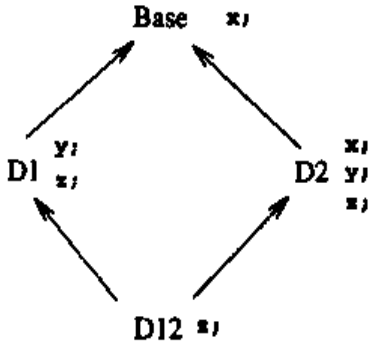


图 5.7 解析中的优先级

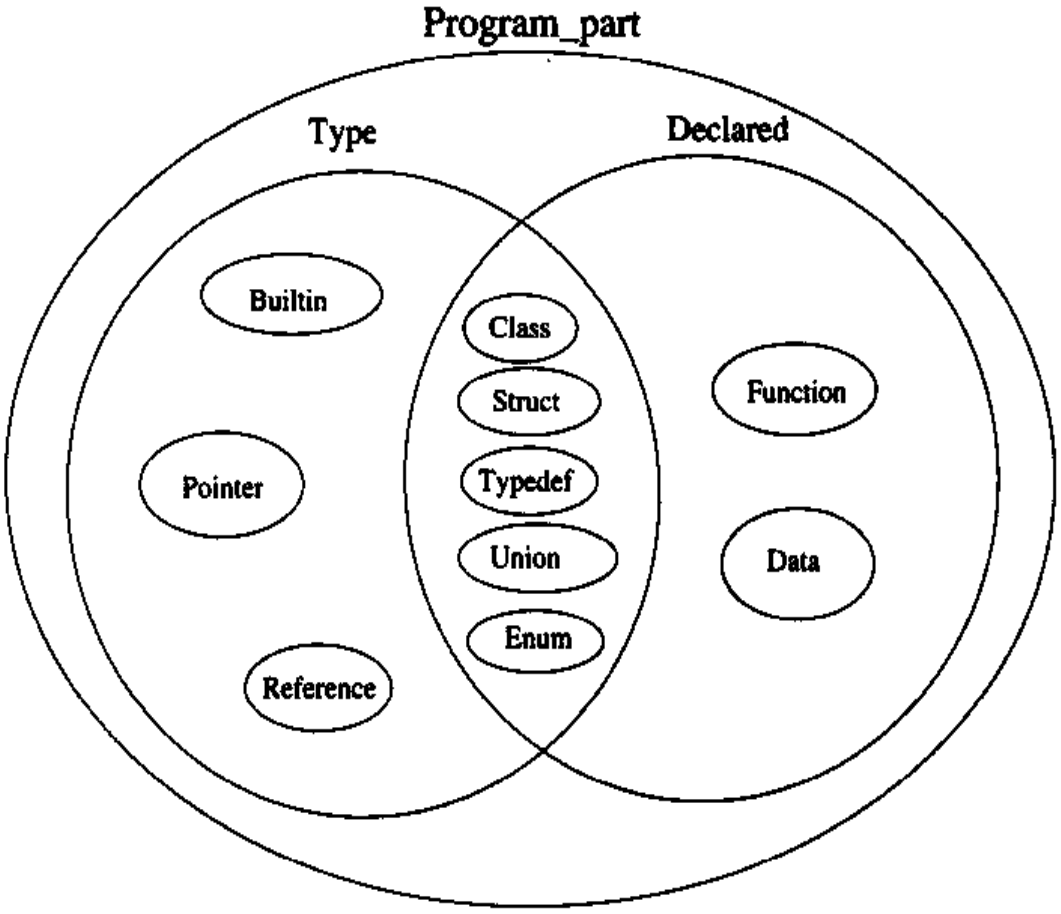


图 5.8 用于描述 C++程序部分的维恩图

图 5.9 中给出的就是我们希望得到的继承树。在 Declared_type 到 Program_part 之间存在着两条不同的路径这个事实也暗示着我们不得不使用虚基类。

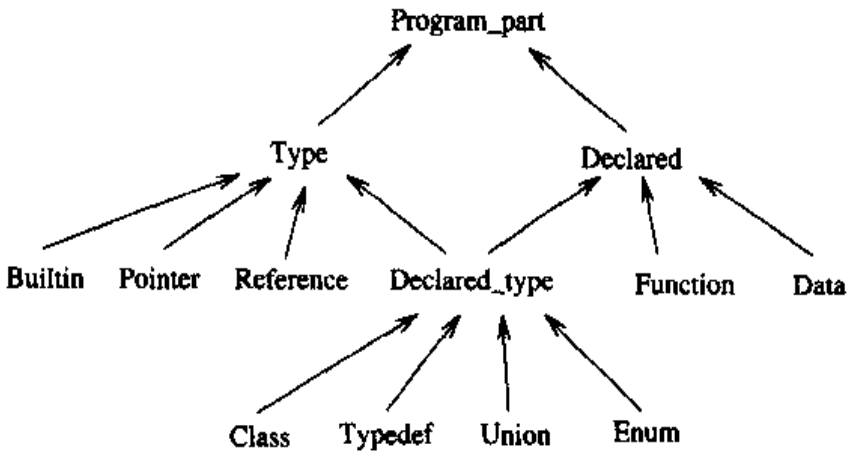


图 5.9 使用了虚基类的继承树

如同我们在本章和第 6 章中将要看到的那样，虚基类会导致事情变得更加复杂。只有在必要的情况下，我们才应该使用它们。但对于我们的这个设计来说，当两个彼此相交的集合同时都是另外一个集合的子集时，使用虚基类来描述这种 is-a 关系就是最恰当的做法。

5.3 一些有关多重继承的细节问题

在本节中，我们将描述一些有关多重继承的细节问题。

5.3.1 名字歧义

当多个基类声明（或继承）了有着同样名字的成员函数时，就很可能产生名字歧义。例如，假设 Type 和 Declared 中都有一个名为 make_String 的成员函数，它会返回一个用来描述对象的 String：

```
class Type {
public:
    String make_String() const;
    //此处忽略细节...
};

class Declared {
public:
    String make_String() const;
    //此处忽略细节...
};
```

那么所有对 Declared_type 进行的 make_String 的调用都会导致歧义。

名字歧义经常（但不是通常）都暗示出我们的设计不是那么尽善尽美。在本例中，我们应该对 make_String 的作用仔细进行考虑，它是不是只对 Type 和 Declared 有意义，还是对所有

程序中的组件都有意义？如果它对所有的程序组件有意义，那么我们应该把 make_String 的声明放到类 Program_part 中去。如果对所有的 Program_part 来说，使用一份简单的 make_String 就可以达到目的，那么我们就可以通过移除其他的 make_String 声明来消除歧义。如果 Type 和 Declared 中对 make_String 的实现不一致，或者 make_String 根本不应该成为 Program_part 的成员，我们也可以通过声明和定义 Declared_type::make_String（它将会把基类中的同名函数给隐藏起来）来修复这种歧义：

```
class Declared_type : public Type, public Declared {
public:
    String make_String() const;
};
```

这样做我们就可以消除掉名字间的歧义。

5.3.2 基类间的初始化

在类声明中有关基类出现的顺序决定了如下三件事情：

- 派生类中的基类部分的构造顺序和它们在声明中出现的顺序相一致。当我们构造 Declared_type 时，Type 部分会先于 Declared 部分被构造出来。（不过虚基类是个特例，我们将在后面对它进行讨论。）
- 派生类中的基类部分将会以它们被构造的顺序相反的顺序被析构。
- 该顺序可以对存储空间的布局细节产生影响，但它不应该影响到程序的作用。

5.3.3 有关虚基类的初始化

如果虚基类中不存在缺省的（不需要参数的）构造函数，那么它就必须在每个派生类中都被初始化。我们将在下一章详细讨论这个问题。

5.3.4 指定存取类型

当声明一个派生类时，我们应该为每个基类都指定相应的存取类型（公有、私有或保护型）。如果我们没有为它们指定相应的存取类型，那么对于声明为 class 的派生类来说，缺省的存取类型就是私有的；对于 struct 来说，缺省存取类型为公有的。

```
class Base1 {};
class Base2 {};

class Derived : public Base1, Base2 {
};
```

即使我们自己清楚：Base2 是一个私有的基类；但那些阅读我们代码的人可能并不一定能知道这一点。将代码改为如下形式会让人更容易理解它一点：

```
class Derived : public Base1, private Base2 {
};
```

5.4 小结

- 只有在派生类包括多个基类，并且这多个基类间没有继承关系时，我们才应该使用多重继承。
- 使用虚基类来避免在同一对象中出现多份基类子对象。
- 基类对象的构造顺序和它们在类声明中（而不是在构造函数的定义中）的顺序一样。
- 为每个基类明确地指定存取类型。

5.5 问题

1. 图 5.10 中是一幅关于基类 A、B 和 C 的维恩图。请写出关于派生类 AB 和 BC 的声明。

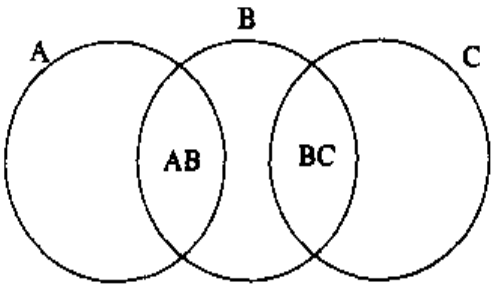


图 5.10 用于描述三个互相相交集合的维恩图

2. 图 5.11 中是一幅关于基类 A、B 和 C 的维恩图。请写出关于派生类 AB、BC、AC 以及 ABC 的声明。

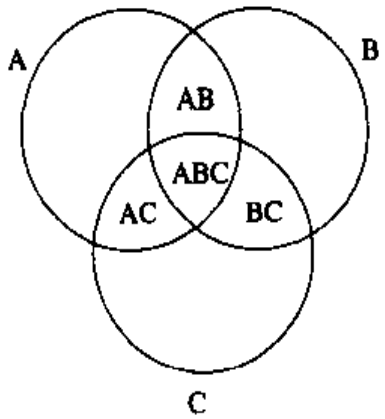


图 5.11 用于描述三个彼此相交集合的维恩图

3. 请为一个有着虚基类的继承架构编写赋值操作符，它将只对虚基类进行一次复制。我们是否值得为它能节约下的那些运行时间来添加额外的复杂性呢？是否还存在着其他原因让

我们只应该对虚基类部分进行一次复制呢？

4. 虚基类中的操作是否可以有（不仅仅是改变对象状态的）副作用？提示：上一题的答案会对这题有影响。

5. 5.2节“回顾：虚基类”中所述的优先级规则对那些没有使用虚基类的多重继承是否也同样有效呢？

6. 请设计出一个可以反映出下面关系的类继承架构：

“X要么是Y，要么就是Z”

在此我们假设Y和Z的派生集之间没有交集。

第 6 章

考虑继承的设计

在面向对象设计的社区中流传着这样一个神话：

如果我们使用了面向对象的设计方法，我们就可以使用其他人所编写的任意类，并通过将它们作为我们类的基类来实现类似的工作。

然而实际上事情并非如此。如果类并不是为继承所设计的（也就是说，类在被设计时就有着这样的意图：该类在将来会被其他类所继承），而我们仍强行让它来作为我们的基类，那么我们就很可能会陷入困境。为了理解为什么这样的继承不能达到我们的期望，我们需要重温一下类向外提供的两种接口：一种面向用户；另外一种则面向它的派生类。

6.1 被保护的接口

依照 C++的语言规则来说，类的面向用户的接口包括它的公用成员；面向派生类的接口则同时包括它的公用和保护型成员。但是我们在第 1 章中学到的知识在此同样也适用：由类所提供的抽象模型要比它成员的类型签名和存取类型都重要。一个不能提供让人容易理解并详细记录的抽象模型的类的用途也不会太大。

一个被设计为不能被继承的类同样也可以向其用户提供一个很好的抽象模型，但它向其派生类所展示的抽象模型则完全是随机构造的！这个随机接口的可用性通常都是很小的。

应该强调的是，我们在此所谈论的是如何设计一个将要被其他拿去当作基类使用的类。很显然，如果我们是要被卖出去的类库的所有者，那么我们就可以通过修改相应的基类来修正我们可能碰到的所有问题。在这种情况下，对不恰当的继承的担心在此没有意义。问题只出在用户对于需要被修改的类没有所有权的时候。

我们先来看看导致不恰当的继承的几种原因。

6.1.1 基类中的成员函数不是虚函数

当派生类需要重写的函数在基类中并没有被声明为虚拟时，我们就很容易得到不恰当的继承。例如，如果第 4.1 节中 Vehicle 类的作者没有将 accelerate 这个成员函数声明为虚拟的，那么通过 Vehicle*所调用的所有 accelerate 在编译时就被绑定到 Vehicle 类中的那个 accelerate 上

面去了。

通常都需要被声明为虚拟的那个函数就是析构函数。如果基类中的析构函数不是虚拟的，那么通过一个指向基类的指针来进行的删除动作将不能调用到正确的那个派生类中的析构函数：

```
class Vehicle {
public:
    Vehicle();
    ~Vehicle(); // Oops,不是虚函数
};

class Land_vehicle : public Vehicle {
public:
    Land_vehicle();
    ~Land_vehicle();
};

class Tank : public Land_vehicle {
public:
    Tank();
    ~Tank();
};

main() {
    Vehicle* vp = new Tank();   // 调用的是Tank::Tank
    delete vp;  // 调用的是Vehicle::~Vehicle
}
```

在上例中，Tank的构造过程很正常：我们调用了Tank的构造函数，它会去调用Land_vehicle中的构造函数，这个构造函数又会去调用Vehicle的构造函数。然而，当我们删除vp时，程序中绑定的是Vehicle的析构函数——而Land_vehicle的析构函数则不会被调用。

如果我们希望其他人来继承我们的类，请确保该类的析构函数是虚函数，即使这个析构函数本来不需要被声明也应该这么做。

6.1.2 基类中的私用成员应该被声明为保护型成员

派生类有时也可能需要访问基类中的私用成员。例如，假设在String类的构造函数中，我们使用了new来控制它的字符串数据：

```
class String {
private:
    char* rep;
public:
    // 此处忽略细节...
```

```
    String(const char* = "");
    ~String();
};

String::String(const char* cp)
: rep(new char[strlen(cp)+1] {
    strcpy(rep,cp);
};

// 复制构造函数和赋值操作符实现细节差不多

String::~String() {
    delete [] rep;
}
```

现在假设我们希望创建一种新的 String，它的功能和其他 String 一样，只不过现在它可以将数据存储到磁盘上去（可被持续化）：

```
class Pers_string : public String {
// ...
```

从概念上来说，这个抽象模型并没有任何问题：Pers_string 也是 String，只不过它可以被持续化而已。然而，当我们关注到 String 的实现细节时，我们很难清楚地找到实现 Pers_string 的方法。我们在 String 的构造函数调用了 new 操作符来初始化私用成员 String::rep，这就说明我们必须将 String 中的字符串放置到内存中（而不是磁盘上）。由于 rep 是一个私用成员，我们也就没有办法来避免这个问题。

6.1.3 非虚基类

另外一个问题发生在“不是为继承所设计的类却被用于多重继承”的时候：

```
class Vehicle {
// ...
};

class Land_vehicle : public Vehicle {
// ...
};

class Sea_vehicle : public Vehicle {
// ...
};
```

假设有人希望创建一种新的交通工具 Amphibious_vehicle（两栖交通工具），它既是 Land_vehicle，也是 Sea_vehicle：

```
class Amphib_vehicle :
    public Land_vehicle,
    public Sea_vehicle {
// *...*
};
```

显然，上面的代码不能正常工作。由于 Vehicle 不是 Land_vehicle 和 Sea_vehicle 的虚基类，我们在 Amphib_vehicle 中就将得到两个 Vehicle 的子部分：一个来自于 Land_vehicle，一个来自于 Sea_vehicle。只有当 Vehicle 被声明为虚基类时：

```
class Land_vehicle : public virtual Vehicle {
// *...*
};

class Sea_vehicle : public virtual Vehicle {
// *...*
};
```

上面的多重继承才可以正常工作。

6.1.4 基类中的假设

在基类的实现中可能存在着一些假设：它们对于基类来说是正确的，但对于派生类来说则不一定。例如，我们在基类中可以假设某些信息是非必须的，或者在基类中有着某些不变量（它们并不是被记录的抽象模型中的部分）。

6.2 我们的设计是否应该考虑到继承？

是不是所有的类在设计时都必须考虑到继承呢？或者进一步说，我们是否应该把所有的成员函数和基类都声明为虚拟的呢？在此我们应该从如下两个方面来考虑：正确性和性能。

6.2.1 正确性

我们将成员函数声明为虚函数是否会带来错误的结果呢？也就是说，假设我们在派生类中重写了该函数，我们是否还会期望在通过指向基类的指针调用该函数时调用到的是基类中的那个函数呢？

```
class Base {
public:
    void foo();
};

class Derived : publice Base {
public:
```

```
    void foo();
};

main() {
    Derived d;
    d.foo();     // 调用的是 Derived::foo
    Base b(d);
    b.foo();     // 调用的是 Base::foo
}
```

在程序中故意使用这样的做法是一种很不好的行为。继承的所有观念都来自于只有对象（而不是调用者）才了解特定操作的实现细节。一个依赖于基于操作的调用方式来选择不同实现的程序，它使用类型系统的目的也只是用来控制程序的实现（而且使用的是那种很难以捉摸的方式来实现这一点）。如果我们假设程序不应该依赖这种行为，那么将函数声明为虚拟的也就不会导致问题的出现。

那么对于虚基类来说情况又如何呢？将一个公有基类声明为虚基类在概念上来说也不会有什么错误。如我们在第 5 章所讨论的那样，is-a 只能有存在与否两种情况，在两个类中包含两份同样的 is-a 关系毫无意义。然而，我们很容易就可以构造出一种情况，在这种情况中，将基类声明为虚拟的会破坏已有程序的功能！此时问题出在构造函数中，当存在虚基类时：所有的虚基类都会在最外层的派生类中被构造。

考虑如下情况：

```
class Vehicle {
public:
    Vehicle(double x = 0.0, double y = 0.0);    // 位置
};

class Land vehicle : public Vehicle {
public:
    Land_vehicle(double x = 0.0, double y = 0.0);
};

Land_vehicle::Land_vehicle(double x, double y)
: Vehicle(x,y) {
    // 忽略其他细节
}

class Tank : public Land_vehicle {
public:
    Tank(double x = 0.0, double y = 0.0);
};

Tank::Tank(double x, double y)
: Land_vehicle(x,y) {
```

```
    // 忽略其他细节
}
```

上面的代码很容易理解。在 Tank 构造函数中的参数指定了 Land_vehicle 构造函数中的开始位置，而在 Land_vehicle 构造函数中的参数又被传递给了 Vehicle 的构造函数。

现在，我们来修改一下这个例子，使得 Vehicle 是 Land_vehicle 的一个虚基类：

```
class Vehicle {
public:
    Vehicle(double x = 0.0, double y = 0.0);   // 位置
};

class Land_vehicle : public virtual Vehicle {
public:
    Land_vehicle(double x = 0.0, double y = 0.0);
};

Land_vehicle::Land_vehicle(double x, double y)
: Vehicle(x,y) {
    // 忽略其他细节
}

class Tank : public Land_vehicle {
public:
    Tank(double x = 0.0, double y = 0.0);
};

Tank::Tank(double x, double y)
: Land_vehicle(x,y) {
    // 忽略其他细节
}
```

现在你能看出因为这种改变所带来的 bug 吗？假设已有的代码如下：

```
Tank t(1.0, 2.0);
```

它实际上会创建一个有着初始位置(0.0, 0.0)的 Tank 对象！这是因为对于虚基类 Vehicle 的初始化是在最外层的派生类（在本例中为 Tank）中进行的。由于 Tank 的构造函数并没有明确地初始化 Vehicle，所以程序就自动调用了 Vehilce 的缺省构造函数。在 Land_vehicle::Land_vehicle 中对 Vehicle 的初始化在本例中实际上并没有被执行——因为 Land_vehicle 并不是最外层的派生类。

要想修正这样的 bug，我们就必须在每个构造函数中都对虚基类进行初始化：

```
Tank::Tank(double x, double y)
: Land_vehicle(x,y),
  Vehicle(x,y) {
```

```
    // 忽略其他细节
}
```

如同我们所见到的，将一个基类改为虚基类的变化还是很大的：它会对所有继承自它的派生类都产生一个强制要求——在构造函数中明确地初始化它。

如果虚基类中没有提供构造函数，或者是只有一个缺省的（不需要参数的）构造函数，由于缺省情况下的行为也就是正确的行为，所以我们不会碰到任何问题；常见的情况就是基类是一个抽象类并且没有成员数据。如果它没有提供缺省的构造函数，一旦用户忘记对虚基类进行初始化，编译器就会为我们给出一个编译时的错误。我们应该避免的是这种情况：在虚基类中同时存在着缺省构造函数和一个带有参数的构造函数。此时，派生类的设计者就需要决定是要明确地对每个虚基类进行初始化，还是使用虚基类中的缺省构造函数来在暗中对它进行初始化。

6.2.2 性能

设计时考虑到继承会对性能带来什么样的影响呢？

将一个函数声明为虚拟的会带来一些开销。每当该函数被调用时，对象必须去检测到底是那个版本的函数被调用。在大多数的架构下，每次调用的开销都只是很少的一些指令。

将函数声明为虚函数同时也会因为该函数不能内嵌而导致一定的开销。如果在编译时编译器断定应该执行的代码究竟是哪部分代码，那么它也就不能将成员函数扩展为内嵌函数。

对于小函数来说，这两方面的考虑都很重要，此时调用函数的时间和它的执行时间相比比较可观。对于大的函数来说，所有函数调用的开销的改动相对函数的执行时间来说根本就是微不足道。所以，将一个小函数声明为虚函数可能会带来很大的性能影响，而将一个大函数改为虚函数则不会对程序的执行效率带来可测到的影响；如果我们对此还有疑问的话，那么对于中等和大规模的函数来说，请不要犹豫，将它们改为虚函数就可以了。

作为一个实验，我构造了一个 String 类，它的底层使用了 char*来表示 String 对象所包含的字符串。它里面的所有成员函数的规模都很小（1 到 5 行）。然后我又构造了一个程序用来对数十个 String 所组成的数组进行排序。在不同的平台上面，我记录了四种不同版本的程序的运行时间：在一个版本中的所有成员函数都被静态绑定，并且还都被扩展为内嵌函数；一个版本中的所有成员函数都是被静态绑定的，但它们没有被扩展为内嵌函数；还有一个版本中的成员函数（除构造函数外）都是被动态绑定的（虚函数）；最后那个版本中使用了虚函数，并且 String 类中还继承了一个无用的虚基类。对于所有的版本，在编译时我都把优化给打开了。表 6.1 中就是我得到的结果：

表 6.1 对 String 数组进行排序所使用的时间

平台 编译器	内嵌函数	非内嵌函数	虚函数	虚基类
SUN SparcStation 10 USL C++ 3.0	1	1.05	1.07	1.10

续表

平台 编译器	内嵌函数	非内嵌函数	虚函数	虚基类
SGI 4D/380S USL C++ 3.0	1	1.06	1.08	1.13
IBM 386SX Borland C++ 3.0	1	1.10	1.13	1.21

设计时考虑到继承会给大量使用该类的程序带来很大的影响。（记住：上面的这些数字来自于对整个程序进行运行得到的结果，而不是仅仅来自于 String 类的实现代码。）除非它可以带来某种好处，否则我们不应该使用这样的代码。从另一方面来讲，对性能的冲击并没有达到让我们一定要修改设计以避免使用虚函数和/或虚基类的地步。如果使用虚基类和虚函数可以让我们的设计更清晰，代码更容易被理解和维护，那么这些性能上的损失也是值得的。

注意：相比“非内嵌函数”和“虚函数”之间的差别来说，“内嵌函数”和“非内嵌函数”的差别总是要大一些。对我所使用的测试机器和编译器来说，动态绑定的最大开销并不在于运行时分发所带来的执行时间，而是来自于“被动态绑定的函数不能被内嵌”这个事实。

当然，所有的这些测试都应该放到一个大的环境中进行。毫无疑问，如果我对自己编写代码的方式进行一些调整，我将得到一些不同的结果。但对大多数现实中的程序来说，“虚函数和虚基类只会带来适量的性能损耗”这个结论总是正确的。

6.2.3 设计时就考虑继承会有一定的开销

我们从上面的讨论中可以发现：设计时就考虑继承会带来一些可观的性能开销，同时也会带来一些显而易见的好处。将函数声明为虚函数带来的运行时开销根据函数的大小和硬件架构的不一样也会变动得很大。将成员函数改为保护型而不是私用的会让我们难以（或不可能）在不破坏现有的派生类的情况下对实现进行修改。虚基类会对每个派生类的每个构造函数都提出需求。如果不需要使用到上述特性，我们也就可以避免这些开销。

我们对上面的讨论所做出的决定也是类接口的一部分，它也必须（和接口中的其他部分一样）被很好地理解、记录，并且不会通过不兼容的方式被改变。

6.3 一些为继承所做的设计的例子

假设我们希望我们的类可以被其他人所继承。我们就应该在开始时就理解和记录下派生类所表示的抽象模型。和类必须向其用户提供一个特殊的抽象接口一样，被用来继承的类也必须向其派生类提供一个特殊的抽象接口（这个接口可能会不同于面向用户的那个接口）。派生类的职责必须被很好地记录下来并被人所理解。

我们不应该假设面向派生类的接口可以向外暴露出某些实现的细节。相反，我们应该像对用户隐藏实现细节一样尽量对派生类隐藏我们类的实现细节。尤其是当我们可以使用保护型函数来作为面向派生类接口时，我们就应该像避免使用公用数据一样，尽量避免使用保护型数据。保护型数据会使得派生类得以了解到我们类的部分实现（参见 2.3 节中的讨论）。虽然出于性能上的考虑这样做可能是必要的，但我们至少还是应该尽量通过提供保护型存取函数来避免使用保护型数据。

再次重申，我们在此讨论的是：如何设计可能会被其他人继承的类。如果我们拥有所有的派生类，那么使用保护型数据将是一种很完美的解决方法。但如果我们设计的是要给其他人从中派生的类，保护型数据就不再适合了。

当我们在设计将要被其他类继承的类时，我们应该在类中声明和定义一个虚的析构函数。

我们再来看其他的一些为继承而设计的例子。

6.3.1 抽象基类

“最纯粹的”为继承所做的设计得到的将是一个抽象基类。抽象类不需要向外提供什么功能；它们只是被用来指定一个接口。实现该抽象模型的职责则被交到派生类中去实现了。

例如，我们有这么一个用来说明简单的字符串抽象的基类：

```
class Abstract_string {
public:
    // 不存在着构造函数
    virtual     ~Abstract_string() {}
    virtual char operator[](int) const = 0;
    virtual int  length() const = 0;
    virtual void output(ostream&) const = 0;
    virtual void dump(char*) const = 0;
};

inline
ostream&
operator<<(ostream& o, const Abstract_string& as) {
    as.output(o);
    return o;
}
```

由于该类并没有实现，所以我们也就不需要为它提供构造函数。然而，我们还是需要声明和定义一个虚的析构函数，以此来确保对 Abstract_string*的删除会调用到正确的派生类的析构函数。这也是一种“可以没有构造函数但却有析构函数”的情况，它可以很好地体现我们的想法。在 dump 函数中，我们会将字符串中的字符复制到调用方提供的缓冲中，在 3.5.2 节中，我们也使用过一个和它类似的函数。

请注意我们对输出操作符的定义。从抽象模型来看，将 operator<<定义为 Abstract_string

的一个虚函数是个不错的想法，但我们却连将它定义为类中的一个成员函数都做不到：因为<<操作符是被一个 ostream&对象（它的最左操作数）所调用的。（如果一个重载了的操作符是一个成员函数，那么它必将是其最左操作数的成员。）所以我们定义了一个纯虚的成员函数 output 来进行实际上的输出操作，并且定义了一个非成员的 operator<<来调用该成员函数。我们可以对 Abstract_string 的任意派生类来使用该操作符。

抽象基类为派生类提供了极大的弹性；但是由于它们没有提供（或者说：提供了很少的）实现，这也使得派生类的负担大为增加。

我们可以用如下的方式来使用 Abstract_string：

```
class Unicode_string : public Abstract_string {
public:
    Unicode_string(const char*);
    // 此处忽略细节 ...
};

class Upper_case_string : public Abstract_string {
public:
    Upper_case_string(const char*);
    // 此处忽略细节...
};

int
count_blanks(const Abstract_string& s) {
    int len = s.length();
    int nblanks = 0;
    for (int i = 0; i < len; ++i)
        if(s[i] == ' ')
            ++nblanks;
    return nblanks;
}
```

count_blanks 函数中所有和字符串相关的操作都是通过调用虚函数实现的，所以我们可以让它和如下两种字符串一起使用：

```
main() {
    Unicode_string s1("hello world");
    Upper_case_string s2("foo bar baz");
    int blanks = count_blanks(s1) + count_blanks(s2);
    // ...
}
```

抽象基类也可以向外提供一些功能。例如，我们可以在 Abstract_string 中将纯虚函数 output 替换成一个使用了其他纯虚函数来实现它自己功能的虚函数：

```
class Abstract_string {
```

```
public:
    virtual void output(ostream&) const;    // 现在不再是纯虚的了
    // 其他函数保持不变 ...
};

void
Abstract_string::output(ostream& o) const {
    int len = length();
    char* chars = new char[len+1];
    dump(chars);
    chars[len] = '\0';
    o << chars;
    delete [] chars;
}
```

上面的这种替换得到的结果应该对于所有的提供了有意义的dump和length函数的派生类来说都有效。由于output仍然是一个虚函数，如果我们想要得到一个更快的实现，我们可以选择在派生类中重写该函数。

6.3.2 为派生类提供服务的类

另外一个为继承所做的设计就是用来为派生类提供服务的基类。考虑如下用来管理计数器的基类：

```
class Uc_object {
private:
    unsigned int count_d;
public:
    Uc_object() : count_d(0) {}
    Uc_object(const Uc_object&) : count_d(0) {}
    virtual ~Uc_object() {}
    void increment() { ++count_d; }
    void decrement() { --count_d; }
    unsigned int use_count() { return count_d; }
};
```

请注意上面代码中的复制构造函数。Uc_object中缺省的复制构造函数应该对计数器进行复制；但由于一个刚被构造出的对象的计数器应该为0（3.2.1小节），所以我们为Uc_object定义了一个复制构造函数用来忽略掉它的参数并将count_d设置为0。

由于复制构造函数中的参数在函数体中被忽略，我们就可以在构造函数的定义中忽略掉它的名字。这样做在C++中是合法的，并且也是被提倡的做法；首先它可以明确告诉大家该参数在函数体中将会被忽略，其次它还可以用来抑止编译器产生“……没有被使用”这样的警告信息。

下面的代码就可以用来记录所有派生自Uc_object的类的对象的使用情况：

```
class Widget : public Uc_object {
//此处忽略细节
};

class Thing_with_widget {
private:
    Wdiget* widget;
public:
    Thing_with_widget(Widget* w);
    ~Thing_with_widget();
    // 忽略掉其他细节
};

Thing_with_widget::Thing_with_widget(Wideget* w)
: widget(w) {
    widget->increment();
}

Thing_with_widget::~Thing_with_widget(Wideget* w) {
    widget->decrement();
}
```

上面代码中的 Widget 的所有用户都可以知道它是否被使用，以及它在程序中被使用的个数。顺便提一下：上面代码中 Uc_object 的实现细节对于 Widget 来说并不可见。

在上面代码中，Uc_object 只是被用来对派生类的调用提供服务。基类同样也可以用来主动调用定义在派生类中的（虚）函数。一个关于基类的主动调用的例子就是：修改 Uc_object，使得当计数器被清 0 时它会知道摧毁掉已有的对象：

```
void Uc_object::decrement() {
    if (--count_d == 0)
        delete this;
}
```

由于 Uc_object 的析构函数是虚函数，上面的删除动作会调用到派生类中的析构函数。

当然，上面的做法只有在处理指向 Uc_object 的指针的代码能够正确地调用 increment 和 decrement 函数时才能得到正确的结果。它同时还要求所有的 Uc_object 都必须通过调用 new 操作符来产生——我们应该严禁对一个自动的或者全局的对象调用 delete 操作符！

让基类来为派生类提供服务只是“一个类为另外一个类提供服务”中的一个特例而已。除此之外，它（“一个类为另外一个类提供服务”）还包括：

- 一个类是另外一个类的成员（has-a 关系）：例如，如果 Car 拥有 Engine，那么 Engine 就可以用来为 Car 提供服务。
- 一个类可以调用另外一个类的成员函数（use-a 关系）。如果我们需要访问一条特定的职员记录，我们可以调用 Directory 中的一个成员函数来实现这一点。

当服务会改变整个对象的本质时，基类就应该用来向派生类提供服务（提供为继承而设计来达到这一点）。C++中提供了几种不同的机制来允许基类改变整个对象的状态：

- 基类可以调用在派生类中被重写的虚函数。
- 基类中可以声明 operator new 和 operator delete 成员函数，当派生类对象被创建和删除时我们会调用到它们。
- 一个指向基类的指针经常被用来“向下转型（down cast）”为一个指向派生类的指针。（如果基类是虚基类或者在派生类中存在两个基类子对象时，这种做法不值得信赖。）

我们很难用一个数据成员对象来影响类中另外的成员，也很难用它们来影响整个类对象的创建和摧毁。

下面给出了一些应该由基类提供的服务的例子：

- 对对象进行持续化。
- 当对象被创建或被摧毁时向外产生一条调试信息。
- 维护对象中的计数器（如我们上面给出的 Uc_object 类）。
- 提供一种“聪明的”内存分配方式（operator new），将它用于对象的新建和删除操作中。

注意：上面提到的所有类都应该以某种方式对整个对象进行改变。下面是一些没有影响到整个对象以及不应该由基类来实现的改变：

- 一个新的 Manager 类，它派生自 Employee 并同时也和其下属的 Employee 有着关联。（我们可以这样实现它：将它派生自 Employee，并在它内部增加一个 Employee* s 的列表数据成员。）
- 一个新的 Big_shot_manager 类，它可以用来批准不同的场合界限下的费用开支。（将它声明为一个派生类，并在其内部增加一个表示可以批准的费用上限的数据成员。）

如果我们对此还不能确信，可以尝试先使用组合这种形式。如果它不能达到我们的期望，那么就该考虑使用基类。例如，通过新增一个数据成员来将一个类持续化的这种做法显然是错误的：我们并没有向类中增加一个新的用于持续化的部分，而只是将整个类都持续化了而已。

6.4 结论

当我们在设计时考虑到继承时，我们应该应用那些在其他类中都使用到的数据抽象的基础原则；我们必须意识到此时我们所面向的用户有两类而不是一类。我们应该避免将实现细节暴露给派生类。我们还应该知道其他人使用我们的类的可能方式，并在类中为派生类设计出相应的接口。

6.5 小结

- 每个类都向外给出两个接口：一个面向用户，另外一个面向派生类。

- 对于一个会被用来作为基类的类来说，我们应该同时设计和记录下这两个接口（以及和它们相应的抽象模型）。
- 为继承所进行的设计会带来运行时的性能损耗，并使得抽象模型更加复杂。对于某些类来说，它可能很有用，但对于其他类来说则非如此。
- 所有被用来继承的类都应该拥有一个虚析构函数。
- 虚基类是在最外层的派生类中被初始化的。
- 避免使用保护型数据。
- 将小的、经常被调用的函数声明为虚函数会导致程序的运行速度显著变慢。
- 对于中等和大规模的成员函数来说，将它们声明为虚函数时不要犹豫——这样做并不会给性能带来显著的变化。
- 使用基类来提供影响到整个对象的服务。

6.6 问题

1. 在什么情况下，一个“聪明的”编译器可以对虚函数调用采取静态绑定的方式？

2. 下面的代码试图在字符串被创建时向外打印一条调试信息：

```
Abstract_string:Abstract_string() {     // 错误的做法
#ifndef NDEBUG
    cout << "New String: " << *this << "\n";
#endif
}
```

为什么它不能正常工作？

3. 声明一个虚的内嵌函数是否有意义？如果有的话，那么在什么情况下我们可以声明一个虚的内嵌函数？

4. 在什么情况下，将大函数声明为虚函数会导致性能上的巨大开销？（提示：是否所有的大函数都必须长时间运行？）

5. “设计时考虑到继承会导致额外的工作”这个事实是否来自于 C++的语言规则？如果我们使用不同的面向对象编程语言来做同样的事情时，这个事实是否依然成立？

第7章

模　板

模板（也被称为参数化类型）可以被用来实现不依赖于被操作的对象类型的算法和数据结构。例如，模板 Stack 可以用来描述如何实现一个包含任意对象的栈；一旦我们定义好了模板，用户就可以编写使用 String、整数以及指针的 Stack 的代码。编译器会为我们找出代码中使用的到底是哪种类型的 Stack 并自动生成这些 Stack 类的实现。

模板可以用来告诉编译器如何在编译期合成代码；从这点来看，模板有点类似于宏。与宏相比，模板有着如下的好处：

- 模板的定义看起来和类和函数的定义差不多；宏通常都需要额外的语法来保证预处理器能够正确处理它（如使用\来作为多行宏中的续行符，使用额外的括号来避免可能出现的优先级问题）。
- 模板可以被自动实例化；编译器会检测模板的使用并会为没有被生成的模板实例自动生成合适的代码。而对于宏的用户来说，他们必须知道到底需要展开的是哪个宏，并且还必须确保宏不会对同一参数展开两次。
- 编译器会为模板产生更好的出错诊断信息[1]。在宏中出现的错误很难被发现，这是因为大部分编译器仅仅是检测被调用的宏，而不是到底在宏的哪部分出错。

回顾：模板

当我们编写模板时，我们也就是在为一个由无限个类（对于类模板来说）或函数（相对于模板函数）组成的集合中编写规格说明书。所有的模板都有着一个或多个模板参数，通常它们只是作为类型的占位符出现在模板定义中[2]。

当我们使用类模板时，我们可以为该类型参数指定实际的类型；C++编译器会自动生成使用实际类型的模板实例。（我们将这种行为称为模板的实例化。）

例如，如果模板 Stack 只使用了一个类型参数，那么用户就可以创建和使用类 Stack<int>（由整数组成的栈）、Stack<char*>（由指向字符的指针组成的栈）、或者是 Stack<String>（由

1 译注：所谓的“更好”只是一个相对的概念，实际上模板的出错信息还是很难理解的。

2 译注：在现在比较流行的 template-metaprogramming 中也经常出现使用值来作为模板参数的情况。

String 组成的栈）的对象。

注意：模板中的“类型参数”只存在于编译期；到目前为止，C++中还没有和运行期的“类型变量”相关的概念。

下面是一个类模板的声明：

```
template <class T>
class Stack {
// 私用成员在此被省略
public:
    Stack();
    int empty() const;
    void push(const T&);
    T pop();
};
```

上面的 Stack 模板带有一个类型 T，由于我们使用了 class 来声明它，所以它就是一个类型参数。

当定义模板的成员函数时，关键字 template 和类型参数都必须重复出现在每个定义中；模板参数（而不是关键字 template 或 class）也必须作为函数名[3]的一部分出现在函数的定义中：

```
template <class T>
Stack<T>::Stack() {
    // 此处忽略细节
}

template <class T>
int Stack<T>::empty() const {
    //此处忽略细节
}

template <class T>
void Stack<T>::push(const T& arg) {
    //此处忽略细节
}

template <class T>
T Stack<T>::pop() {
    //此处忽略细节
}
```

当定义模板类的构造函数时，模板参数只需要被指定一次就可以了，如：我们定义中使用的是 Stack<T>::Stack，而不是 Stack<T>::Stack<T>

即使我们使用的是关键字 class 来为模板指示类型参数，模板的实际参数还是可以为任意

3 译注：实际上模板参数只是作为类型名的部分出现在函数名的限定形式中。

的类型——我们并不需要它是一个类：

```
Stack<char*> ptr_stack; // OK
```

由于模板是语言的一部分，而宏不是，所以上述的所有好处也就显得很自然了。

7.1 模板类 Pair

我们首先从构建一个简单的模板类开始。在 C++中，函数的返回值只能是一个对象。如果我们要想使函数返回两个对象，可供我们选择的设计就有着如下两种。

函数可以通过让调用方向函数中传递指针或引用参数来“返回”这两个对象中的一个或全部，但这样做会有一些缺点：

- 调用方必须明确地定义存储返回值的对象。如果产生对象的类中有着一个构造函数但却没有缺省构造函数时，这样做至少会给我们带来编写程序方面的麻烦，严重时还会导致程序出错。
- 如果产生对象的类中有一个构造函数，从函数中“返回”它通常都会导致一个新的对象被构造（构造时使用的是缺省构造函数）。被调用的函数通过赋值操作来改变该对象的值。如同我们在 2.1.3 节中所见到的，用缺省构造函数来初始化一个对象然后再用赋值操作来改变它的值指针做法通常都要慢于直接用正确的值对这个对象进行初始化。
- “函数将返回值存储在它的某个引用参数中”这个事实在函数调用中不是很明显。
- 如果调用方向函数中传递的是一个指向派生类的引用，那么在存储过程中只有对象的基类部分的值会得到改变。这可能会使得对象处于不安全的状态下。

另外一种做法就是让函数返回一个包含这两个返回值的对象。我们可以设计一个模板 Pair<L, R>来做到这一点，然后在我们的程序中直接使用 Pair 而不去显式声明 Pair 中究竟该放置什么样的数据。例如：

```
Pair<String, int> find_highest_salary(const Database&);

void print_highest_salary(Database& db) {
    Pair<String, int> rch = find_highest_salary(db);
    cout << rch.left() << " earns " << rch.right() << endl;
}
```

一个 Pair<L, R>对象中包含了一个 L 的对象和一个 R 的对象：

```
template <class L, class R>
class Pair {
private:
    L   left_d;
    R   right_d;
```

```
public:
    Pair(const L& l_arg, const R& r_arg)
    : left_d(l_arg), right_d(r_arg) {}
    L   left() const { return left_d; }
    void    left(const L& l_arg) { left_d = l_arg; }
    R   right() const { return right_d; }
    void    right(const R& r_arg) { right_d = r_arg; }
};
```

上面的就是 Pair 的所有实现代码了。我们提供了一个构造函数来初始化它的成员，我们没有必要为它提供析构函数。如果 L 或 R 中有着自己的析构函数，那么编译器就会自动为我们的 Pair<L, R>合成一个适当的析构函数；否则的话，Pair<L, R>就没有析构函数。缺省的复制构造函数和赋值操作符的行为都能满足我们的期望，所以我们也不需要在程序中提供它们。

我们也可以设想一个用来包含三个对象的模板类 Triple<L, M, R>；然而，在某些情况下，定义一个新的类会比让函数返回一个（假设中的）Quint<A, B, C, D, E>要更有实际意义一些。

如果我们希望重载模板，会有什么情况出现呢？例如，我们可能会设想模板 Tuple 可以用来接受任意个类型参数。可惜的是，模板类不能被重载，对于一个特定的名字来说，程序中只可以用它来定义一个模板类[4]：

```
template <class L, class R>
class Tuple { /* ... */ };
template <class L, class M, class R>
class Tuple { /* ... */ }; /*编译期错误：Tuple 被重复定义*/
```

7.2 一些有关模板的细节

模板类的两个不同实例间的关系必须被明确地区分开来。Pair<int, int>和 Pair<double, double>是两个完全不同的类；它们之间既不存在继承关系，也不存在着友关系。实际上，我们根本没有办法让同一个模板类所产生的实例类相互成为友类，我们可以这么声明：

```
friend class Stack<int>;
```

但不能这么做（或者其他类似的行为）：

```
friend template<class T> class Stack;
```

我们可以设法让一个特定的类成为一个指定的模板类的所有实例的友类：

```
template <class T>
class Stack {
friend class Stack_impl;
```

4 译注：我们可以使用递归的方式来使得 Tuple 接受任意个模板参数变为可行的，具体的做法可以参考 boost 库中 Tuple 库（http:// www.boost.org/libs/tuple/）。

我们也可以设法让模板类的所有实例类都拥有一个共同的基类：

```
template <class T>
class Stack : public Stack_base {
```

在上面的代码中，只有 Stack 是模板类；Stack_base 是一个非模板类，所有的 Stack 都继承自 Stack_base。

我们可以用模板类来作为其他模板的类型参数：

```
template <class T>
class List {
// ... 此处忽略细节
};

template <class T>
class Set {
// ... 此处忽略细节
};

List< Set<int> > list_of_int_sets;
```

上面的代码中声明了一个链表，它里面的每个元素又是一个 Set<int>。

在模板类中用来为类型参数定界的<和>还会导致一个语法上的小问题。在本例中，我们必须在两个>中插入一个空格[5]：

```
List< Set<int> > list_of_int_sets; // OK
List< Set<int>> list_of_int_sets;       /*语法错误：未预料到的'>>'*/
```

如果我们忽略了这个空格，编译器就会将>>作为一个右移操作符来处理。

7.3 模板的实例化

当编译器检测到我们通过一个特定的类型参数集合来使用模板时，它会为该模板使用这些类型参数实例化出一个模板的实例（实例化的动作就是用这些类型参数去替换模板声明中的模板参数）。然后编译器会编译该实例得到它的目标代码文件并把该文件传递给链接器去使用。

例如，假设我们使用了一个已有的模板 List 来实现我们的 Stack：

```
#include <list.h>

template <class T>
class Stack {
private:
    List<T> list;
```

5 译注：据说在 C++ 0x 标准文档中，这个问题将会得到解决。

```
public:
    Stack();
    ~Stack();
    void push(const T&);
    T pop();
};
```

上面的模板成员定义中使用了已有的 List 类中的成员函数。push 操作会将一个元素插入到 List 的末端，pop 操作则会从 List 中移除掉第一个元素：

```
template <class T>
void Stack<T>::push(const T& elem) {
    list.insert_at_head(elem);
}

template <class T>
T Stack<T>::pop()
{
    return list.remove_from_head();
}
```

如果有用户创建了一个 Stack<int>对象，并对它调用了 push 和 pop，那么编译器将会创建和编译类似于下面代码的实例文件：

```
void Stack<int>::push(const int& elem) {
    list.insert_at_head(elem);
}

int Stack<int>::pop() {
    return list.remove_from_head();
}
```

当模板的实现中使用到了其他模板时，上述的实例化操作也可以迭代进行下去。例如，Stack<T>中使用了 List<T>来作为它的实现的一部分。那么当编译器实例化 Stack<int>时，（在编译完 Stack<int>的成员函数后，）它会发现 List<int>也需要被实例化。有可能 List<int>中也会需要实例化其他的模板……这样我们的实例化操作将会一直迭代下去，直到所有需要被实例化的模板都被实例化了为止——或者直到碰到一个实例化出错的模板为止。

实例化出错发生在实例化得到的文件不能被正确地编译时。出现这种情况的原因可能是模板类的定义中存在着 bug，或者是模板中的代码试图对特殊的类型参数做一些非法的操作（参见 8.7 节）。

7.3.1 实现细节

不幸的是，现有的 ISO/ANSI C++标准文档的草案并没有指定编译器应该如何实现模板的实例化。这意味着我们必须去咨询我们使用的编译器所附带的文档，来确定我们的模板成员函

数应该放在什么地方。

我们应该对编译器如何缓存模板的实例保持密切的关注。在一个大的项目中，我们可能会使用到超过 100 个的模板实例；如果在每次链接时我们都对它们进行重编译的话，用于编译上的时间开销将会变得非常大。一个可靠的编译器应该只对每个实例编译一次并将编译后的目标代码保存在一个缓冲区中，以便在随后的编译中可以重用到这些目标代码。我们也可以提供某种机制来在必要时（例如：模板的定义发生了改变或者它的参数发生了改变）更新这些缓冲区。再次重申一遍：不同的编译器对此的操作都不尽相同，我们必须根据我们编译器的文档来提供相应的机制。

7.4 智能指针

在第 3 章中，我们在对象的实现中使用了句柄；我们在那里所使用的句柄都是一些普通的 C++指针。不过我们也可以用对象来实现句柄；这些对象的行为和指针差不多，但它们还可以提供一些 C++指针所不能提供的功能。

在我们讨论那些新增的功能之前，我们先来看看如何设计一个和指针功能相似的类。我们在这些“智能指针”里面使用了 operator->()来让用户写出和使用常规的 C++指针一样的代码。同时我们还提供了一个用来进行隐式类型转换的函数：

```
class String {
public:
    String(const char* = "");
    int length() const;
    //此处忽略细节
};

class String_ptr {
private:
    String* ptr;
public:
    String_ptr(String* s) : ptr(s) {}
    String* operator->() const { return ptr; }
    operator String*() const { return ptr; }
};

main() {
    String s("hello world");
    String_ptr sp = &s;
    int len = sp->length(); // (sp.operator->())->length()
    String* dumb_ptr = sp;  // (sp.operator String*())
}
```

请注意上面 main 函数中的最后两行。对于 sp->length()的调用实际上调用的是 sp 对象中被重载的->操作符。(注意：sp 是一个对象，并不是一个指针。) String_ptr::operator->()返回的是一个 String*，我们的 length()就是基于这个 String*被调用的。在最后那行中，我们用 String_ptr::operator String*()的返回值来对 String* dumb_ptr 进行初始化。

实际上，大多数的“智能指针”都和它们所指的对象无关。我们可以用模板来实现这样的智能指针：

```
template <class T>
class Ptr {
private:
    T*  ptr;
public:
    Ptr(T* p) : ptr(p) {}
    T* operator->() const { return ptr; }
    operator T*() const { return ptr; }
};
```

在定义了这个模板后，我们就可以使用 Ptr<String>来作为“指向 String 的指针”在我们的代码中使用：

```
main() {
    String s("hello world");
    Ptr<String> sp = &s;
    int len = sp->length(); // 与前面相同...
    String* dumb_ptr = sp;  // 与前面相同...
}
```

由于 Ptr 中的所有成员函数都是内嵌函数，所以上述代码在运行时并不会慢于使用常规 C++指针的代码。当然，如果智能指针还有着其他的额外功能，那么它就可能慢于常规指针，但这种性能上的开销主要来自于这些新增的功能，而不是因为我们对智能指针中的功能进行包装时产生的损耗。

7.4.1 对非自定义类型使用 operator->

我们的 Ptr 模板可以很好地协同自定义类型一起工作，但如果我们希望实例化一个 Ptr<int>，编译器将会给我们如下的错误信息：

```
error: operator->()必须返回一个指向类对象的指针、引用或者类对象
```

编译器在此抱怨无法对 Ptr<int>调用 operator->，这是因为我们调用它的目的只是为了访问 int 中的成员函数——而这显然是不可能的（因为 int 并不是一个类）。如果我们希望提供一个指向非自定义类型的智能指针，我们就需要提供两个版本的智能指针：一个声明有 operator->，另外一个则没有。我们在此可以使用公有继承；在 NCPtr<T>（“非自定义类型的

指针”）中不声明 operator->，这样我们就可以使用任意类型来作为 T 的实参；然后我们再从 NCPtr<T>中派生出 Ptr<T>，我们在它的定义中提供 operator->的声明：

```
// 指向任意类型的指针：
template <class T>
class NCPtr {
private:
    T*  ptr;
public:
    NCPtr(T* p) : ptr(p) {}
    operator T*() const { return ptr; }
};

// 指向自定义类型的指针：
template <class T>
class Ptr : public NCPtr<T> {
public:
    Ptr(T* p) : NCPtr<T>(p) {}
    T* operator->() const { return (T*)(*this); }
};

main() {
    int i;
    NCPtr<int> ip = &i;
    String s;
    Ptr<String> sp = &s;
    int len = sp->length();
}
```

注意：我们在此重写了 Ptr<T>::operator->的实现，我们使用了 NCPtr 中的公用接口，这使得我们可以将数据成员 NCPtr<T>::ptr 继续保持为私用的。

到目前为止，我们的智能指针并不比常规指针“聪明”多少。所以我们现在要给它增加一些额外的功能。

7.4.2 使用了计数器的智能指针模板

在本节中，我们实现了一个智能指针模板用来仿效垃圾收集机制——这里面会用到计数器。所有使用计数器的对象都可以派生自（非模板的）类 Uc_object。对于这种类型的对象分配通常都是通过调用 new 操作符来实现的。所有使用了计数器的对象 T 都可能同时被多个 Uc_ptr<T>所指向。当 Uc_ptr 所指向的对象中的计数器从 1 变为 0 时，该对象就将被删除。（注意：引发删除操作的是从 1 到 0 的这次变迁，而不是计数器为 0 这个事实。）我们之所以这样做是因为：当对象被创建时，我们给它里面的计数器的初始值就是 0，而这个时候对象是不应该被删除的。

在第6章中，我们就定义了基类Uc_object如下：

```
class Uc_object {
private:
    unsigned int count_d;
public:
    Uc_object() : count_d(0) {}
    Uc_object(const Uc_object&) : count_d(0) {}
    virtual ~Uc_object() {}
    void increment() { ++count_d; }
    void decrement() { if (--cound_d == 0) delete this; }
    unsigned int use_count() { return count_d; }
};
```

上面代码中的increment和decrement函数被用来管理计数器的增减；当计数器变为0时，我们会在decrement函数中将对象给删除。

我们可以这样地来实现模板 Uc_ptr：当它被创建、摧毁或值发生改变时，我们就会去调用相应的increment和decrement。由于被指向的对象必须以Uc_object作为其基类，而所有的Uc_ptr都必须指向这个对象，所以我们就可以不用去声明那个不带operator->版本的模板：

```
template <class T>
class Uc_ptr {
private:
    T*  ptr;
public:
    Uc_ptr(T* = 0);
    Uc_ptr(count Uc_ptr<T>&);
    ~Uc_ptr();
    const Uc_ptr<T>& operator=(const Uc_ptr<T>&);
    operator T*() const { return ptr; }
    T* operator->() const { return ptr; }
};
```

在C和C++中，自动指针的初始值是未定义的。而对于我们的Uc_ptr来说，它总会被初始化为一个适当的值（缺省情况下是0）。这样做不仅在实际中来说很有意义，还会促使我们正确地实现赋值操作符和复制构造函数（我们将在下面对此进行讨论）。

上面代码中的两个内嵌的成员函数为我们的模板 Uc_ptr 提供了类似于指针的功能。其他的成员函数则被用来管理我们的计数器，在这些函数中，我们必须仔细检测“从1到0”这种变迁情况。

我们会在构造函数中对计数器进行递增：

```
template <class T>
Uc_ptr<T>::Uc_ptr(T* arg)
: ptr(arg) {
```

```
    if (ptr)
        ptr->increment();
}

template <class T>
Uc_ptr<T>::Uc_ptr(const Uc_ptr<T>& arg)
: ptr(arg.ptr) {
    if (ptr)
        ptr->increment();
}
```

然后在析构函数中对它进行递减：

```
template <class T>
Uc_ptr<T>::~Uc_ptr() {
    if (ptr)
        ptr->decrement();
}
```

另外，在赋值操作符中，我们还需要对“自赋值”情况进行检测：

```
template <class T>
const Uc_ptr<T>&
Uc_ptr::operator=(const Uc_ptr<T>& arg) {
    if (ptr != arg.ptr) {
        if (ptr)
            ptr->decrement();
        if (ptr = arg.ptr)
            ptr->increment();
    }
    return *this;
}
```

现在我们就能够理解为什么我们不能让未初始化的自动指针 Uc_ptr 拥有一个随机初始值了。我们能够对有着随机值的对象进行的惟一有意义的操作就是使用赋值来改变它的值；而这也可能导致程序崩溃——因为 ptr 数据成员拥有的是一个随机值，当它不为 0 时，我们会在 operator=中进行如下的调用：ptr->decrement()。有鉴于此，我们必须在构造函数中对 Uc_ptr 进行适当的初始化。

使用Uc_ptr

现在我们就可以使用 Uc_ptr 来重新实现我们在第 3 章中所说的 String 类了。我们可以用一个带计数器的对象来作为我们的 rep 类：

```
class String_rep : public Uc_object {
private:
    friend  class String;
    char*   data;
```

```
        String_rep(const char* cp);
        ~String_rep();
    // 其他函数忽略 ...
};

String_rep::String_rep(const char* cp)
: data(new char[strlen(cp)+1]) {
    strcpy(data,cp);
}

String_rep::~String_rep() {
    delete [] data;
}
```

然后我们在 String 类中用一个带计数器的指针来指向这个 rep：

```
class String {
private:
    Uc_ptr<String_rep> rep;
public:
    String(const char* cp = "")
    : rep(new String_rep(cp)) {}
    // 其他函数忽略 ...
};
```

为什么我们需要这样做呢？

我们现在就不再需要提供 String 类中的复制构造函数和复制操作符的定义了——因为缺省的版本就已经足够了。对一个 String 进行复制会导致 Uc_ptr 中的复制构造函数被调用，以此来改变 rep 中的值；对 String 进行赋值时，Uc_ptr 中的赋值操作符就会被调用，rep 中的值从而也就得到了改动。上面的操作会自动将 String_rep 中的计数器值给更新。

同样，我们现在也不需要提供 String 的析构函数定义了。缺省的析构函数就可以满足我们的需求了：它会摧毁掉 Uc_ptr 中的数据成员 rep，这也会导致在“没有 String 使用时，String_rep 会被自动摧毁”。

最重要的是，关于计数器的逻辑现在被放在了一个独立的类中，这样我们就可以在随后对它进行很好的重用了。

使用计数器的缺陷

然而，上面给出的方法并不是尽善尽美的。所有使用计数器的对象都必须派生自类 Uc_object（但在问题 5 中我们不希望如此）。我们必须使用 new 操作符来创建它们，否则如果我们使用 delete 来删除它们时就会出现未定义的情况。

那些包含了 Uc_ptr 的循环指向的对象将永远不会被删除：

```
class Item {
private:
```

```
    Uc_ptr<Item> next;
public:
    Item() : next(0) {}
    void point_at(Item* i) { next = i; }
};

void make_cycle() {
    Item* x = new Item;
    Item* y = new Item;
    x->point_at(y);
    y->point_at(x);
}
```

当我们从 make_cycle 中返回时，我们就再也无法访问到它里面的两个 Item。由于它们都被一个 Uc_ptr 所指向，我们也不能对它们进行清理。

最后，如果我们使用了常规的“哑”C++指针，当最后的“智能”指针被摧毁时可能会导致这些指针从此变为空悬指针。

即使这样，这个指针类还是为我们提供了一些好处：如果需要的话，它使得我们可以使用某种垃圾收集机制，并且只有在我们使用了这种机制时，我们才需要为此付出代价。

7.5 作为模板参数的表达式

大部分的模板参数都是类型，但我们也可以使用常量表达式来作为我们的模板参数：

```
template <class T, int SIZE>
class Fixed_array {
// ...
};

Fixed_array<int, 5> intray;// 包含有 5 个元素的数组
```

使用表达式来作为模板参数的情况比较罕见，因为我们可以通过向构造函数中传递表达式参数来做到同样的事情：

```
template <class T >
class Fixed_array {
public:
    Fixed_array(unsigned size);
// ...
};
```

将表达式作为参数传给构造函数而不是模板有着如下的好处：

- 构造函数的参数可以是非常量表达式；而模板参数只能接受常量表达式。
- 当表达式被传递给构造函数作为参数时，它只需要出现一次——只有在对象初始化时

我们才会使用它。如果将表达式作为参数传给模板，那么在每个涉及到模板参数的地方表达式都会被提到。我们也可以使用 typedef 来避开这种不便，但这又会导致每次模板被实例化时都产生一个新名字。

- 由于所有的 Fixed_array<T>对象都具有同样的类型，我们就可以编写函数来接受一个指向 Fix_array<int>（在此选用 int 只是为了举例方便）的指针或引用作为其参数，并让该函数可以工作于任意大小的固定数组上。而要让 Fixed_array<T, int>能够满足同样的条件，就必须要求所有的 Fixed_array<T, int>都派生自同一个基类。

只有当类比较小且我们对性能要求很高时，我们才会在模板中使用表达式参数。把表达式作为模板参数会使得某些实现变快，在内嵌函数中使用这些参数时这种变化显得尤为明显。例如，对比如下两种可能的 Fixed_array 的实现：

Size 作为模板参数被使用：

```
template <class T, unsigned SIZE>
class Fixed_array {
    T data[SIZE];
public:
    T& operator[](int off) { return data[off]; }
    unsigned size() const { return SIZE; }
};
```

Size 作为构造函数的参数被使用：

```
template <class T>
class Fixed_array {
    T* data;
    unsigned size_d;
public:
    Fixed_array(unsigned sz)
    : data(new T[sz]), size(sz) {}
    ~Fixed_array() { delete [] data; }
    T& operator[](unsigned off) { return data[off]; }
    unsigned size() const { return size_d; }
};
```

上面的代码中对 Fixed_array<T, int>的创建和摧毁要稍快一些——它里面没有调用 new 操作符。对它们进行复制同样也要快一些。例如，在一个 16MHz 的 Sparc 工作站上，创建一个有着 100 个元素的 Fixed_array<int>需要大概 20 微秒；而创建一个同样大小的 Fixed_array<int, 100>则根本不需要任何时间。

在实际中，这种通过使用模板的表达式参数而获得的性能上的提升和由此受到的语义限制（尤其是“表达式必须是一个编译期的常量表达式”这个限制）相比并不合算。

7.6 模板函数

到目前为止，我们谈论的都是模板类；我们同样可以声明和定义模板函数（参见下面的评论）。与模板类不同的是，我们在调用模板函数时并不需要为它指定模板参数。由于模板类型参数必须在参数类型中被提及，C++中的函数重载机制就可以为我们找出实际参数的模板类型参数来。例如：

```
template <class T>
void swap(T& a, T& b) {
    T temp(a);
    a = b;
    b = temp;
}

main() {
    String s("Foo"), t("Bar");
    swap(s, t);
}
```

通过传递给 swap 的参数 s 和 t，编译器就知道我们希望调用的是 swap(String&, String&)。

回顾：模板函数

模板函数是用来声明一系列函数的模板。例如，下面的模板函数会将两个有着同样类型的值进行互换（请注意函数定义中我们并没有使用常量引用的形式）：

```
template <class T>
void
swap(T& a, T& b) {
    T temp(a);
    a = b;
    b = temp;
}

main() {
    int x = 5;
    int y = 7;
    swap(x, y); // 现在 x == 7, y == 5
}
```

在函数的参数列表中，我们必须给出所有的类型参数信息[6]：

```
template <class T>
void foo(T*);   // OK: 我们在参数列表中提到了 T
template <class T>
void bar(); // 编译错误：T 没有出现在参数列表中
```

对于模板参数来说，不存在隐式的类型转换：

```
long long1, long2;
int int1, int2;
main() {
    swap(long1, long2); // 调用的是 swap(long&, long&);
    swap(int1, int2);   // 调用的是 swap(int&, int&);
    swap(long1, int2);  // 编译错误：找不到合适的 swap
```

上面代码中的最后一个调用会出错的原因就是：模板 swap 要求它的两个参数都指向同样的类型。

7.6.1 模板函数 sort

我们可以使用上面的模板函数 swap 来构建一个模板函数 sort：

```
template <class T>
void sort(T*v, int n) {
    for (int gap = n/2; gap > 0; gap /= 2)
        for (int i = gap; i < n; i++)
            for (int j = i - gap;
                j >= 0 && v[j+gap] < v[j];
                j -= gap)
                    swap(v[j], v[j+gap]);
}

extern int_ray[20];
extern String string_ray[10];

sort(int_ray, 20);      // 调用的是 sort(int*, int);
sort(string_ray, 10);   // 调用的是 sort(String*, int);
```

它可以用来对所有提供了 operator<的对象进行全排列。

6 译注：原文这种说法是不完全正确的：我们可以不用在函数的定义体的参数列表中不给出所有的类型参数，但只要在函数的调用语句中补足这些信息就可以了。如：

```
template <class T> foo();

foo<int>();
```

上面的代码就可以正常工作。

7.6.2 代码冗余

使用模板函数很容易得到大量的冗余的代码。模板必须精确匹配这条规则意味着：在通常情况下，我们需要为每个不同的类型集合产生一份独立的模板实例；在模板的实例化过程中不会出现隐式类型转换（如从 Derived*转换至 Base*）。

我们可能会试图通过将常用代码放到非模板的函数中去来避免这种代码冗余。但这样做的难度要大于我们所想到的难度。类中还有着一些直接支持这种功能提取的特性（如：虚函数，从派生类到基类间的隐式类型转换等），但对于函数来说则不存在着这样的语法特性。

例如，我们回到上面给出的模板函数 sort。作为第一步，我们可能会试图对该函数的结构进行重新编排，尽可能地在模板中出现和 T 相关的逻辑。实际中和 T 相关的代码包括对数组进行索引访问（这依赖于 sizeof(T)），元素间的值互换操作（依赖于 T 的赋值操作），以及对 v[j]和 v[j + gap]进行比较（依赖于 T 中的<比较关系）：

```
template <class T>
void sort(T*v, int n) {
    for (int gap = n/2; gap > 0; gap /= 2)
        for (int i = gap; i < n; i++)
            for (int j = i - gap; j >= 0; j -= gap) {
                if (!v[j+gap] < v[j])) // 和T相关
                    break;
                swap(v[j], v[j+gap];    // 和T相关
            }
}
```

然后我们就试图将函数划分为两部分：一部分是由和 T 相关代码组成的模板；另外一部分则是一个和 T 无关的函数：

```
template <class T>
int
sort_hack(T& first, T& second) {
    if (!(second < first))
        return 1;
    swap(first, second);
    return 0;
}

void sort_common(void* v, int n) {
    for (int gap = n/2; gap > 0; gap /= 2)
        for (int i = gap; i < n; i++)
            for (int j = i-gap; j >= 0; j -= gap)
                sort_hack(?);   // Oops，现在该怎么办呢?
}
```

但这样做还会有一个问题：我们无法在非模板的函数中调用模板函数。由于模板函数的类

型会随着场景的不同而不同，我们也就无法将它的函数指针作为参数传给这个非模板函数。对于对象来说，只要它们都派生自同一个基类，我们就可以在不同的函数调用中使用不同类型的对象作为参数；而对于函数来说不存在相似的特性。

至此，我们可能会考虑构建一些私用的对象来使得这一切能够工作。然而一个更好的主意就是放弃这个念头并接受由于多份 sort 函数带来的空间开销。因为代码冗余带来的内存开销几乎总是小于使用好几页令人费解的代码来实现一个简单的排序操作所带来的开销。

这意味着什么呢？我们必须谨慎地使用模板函数。它们最适合的地方是那些小函数，因为小函数不会带来太多的空间开销。

7.7 小结

- 当用于处理对象的逻辑和对象本身无关时，我们最好使用模板。
- 模板实例间的关系必须被明确地用程序注明。
- 当模板类同时也是模板参数时，请在两个>间加入空格以避免语法错误，如：

```
List< Set<int> >
```

而不是

```
List< Set<int>>
```

- 模板的实例化细节和我们使用的具体编译器相关。
- 通过提供 operator T*和（对于指向自定义类型对象的指针来说）operator->，我们就可以实现智能指针模板。
- 在模板中使用表达式作为参数会使得使用模板的难度变大，但对于极度关注性能的类来说，这种不便还是值得的。
- 尽量减小模板函数的规模。

7.8 问题

1. 使用 Pair 模板来从函数中返回两个值与创建一个新的类来实现同样的操作这两者之间有什么权衡？请从程序的易理解性，易使用性，程序占用的空间和执行的时间这几个方面来考虑。

2. 设计和实现一个“智能指针”类，它的行为基本上类似于常规指针，惟一的区别在于：对这样的一个空指针进行间接引用会导致程序向外打印一条错误诊断信息并调用 abort。

3. 如果 Foo 不是派生自 Uc_object，那么使用 Uc_ptr<Foo>会出现什么情况呢？

4. 我们的 Uc_ptr 类对常量对象来说不能正常工作。为常量对象提供一个使用计数器的指针是否有意义？如果有的话，那对于这种指针来说，必须支持什么操作？

5. 设计一个新版本的 Uc_ptr，它里面并不要求被指向的对象派生类自同一个基类。将这个新的版本和前面我们给出的版本相比较，它们间的权衡是什么？

6. 除了使用模板函数 swap，我们也可以试着编写一个（非模板）函数，它接受 void*作为其参数，并对它们所指向的内存块进行互换：

```
#include <memory.h>
void
swap(void* left, void* right, int length) {
char* sav = new char[length];
memcpy(sav, left, length);
memcpy(left, right, length);
memcpy(right, sav, length);
delete [] sav;
}
```

这是不是一个好的做法？如果不是，为什么？

7. 为什么 Fixed_array<T, int>要好于普通的数组？

8. 编写一个模板函数 sort，它里面使用的不是<，而是一个用户提供的比较函数。请同时给出两个版本的 sort：一个将该比较函数作为模板的表达式参数，另外一个则将它作为常规的参数使用。请比较这两个版本之间的运行速度的差异。如果该比较函数是内嵌函数时会不会有所不同？

9. 在第 7.6.2 的代码中，为什么我们要这么编写：

```
if (!(v[j+gap] < v[j]))
```

而不是：

```
if (v[j+gap] >= v[j])
```

10. 设计和实现一个“可逆指针”的智能指针模板。一个可逆指针的行为和常规指针一样，但它还可以从被指向的对象中找到所有指向它的指针。

第 8 章

模板的高级用法

在本章中，我们将会使用模板来构建用于提供基础的数据结构链表和集合的容器类。然后我们还会就如何对模板的设计和实现进行细化。

8.1 使用了模板的容器类

容器类是一种实现了某种数据结构，可以用来“容纳”其他对象的类。容器包括数组、链表、堆栈、集合、口袋以及目录等。由于管理容器的逻辑通常都和容器中的内容无关，所以用模板来实现容器是最适合不过的了。

一旦我们定义或使用了某个容器，我们就必须注意如下两点：

- 容器中的元素是同种类型的呢？还是不同类型的呢？
- 容器中的元素是具有引用语义呢？还是具有值语义？

8.1.1 同型和异型？

如果容器只能容纳一种类型的对象，那么它就是同型的。如果它可以容纳不同类型的对象，它就是异型的。在 C++中，异型容器经常被用来容纳派生自同一基类的对象；我们可以在程序中对容器的内容进行遍历，通过基类提供的成员函数对元素进行访问。

8.1.2 引用语义和值语义？

容器中所容纳的是否是对象本身，还是对象的引用？包含有对象的容器就具有值语义。具有值语义的容器可以有着如下的行为：

- 同一对象不可能同时存在于多个容器中（对象不能被分享）。
- 对象是被复制到容器中去的，如：

```
Container_with_value_semantics<Thing> container;
Thing t;
container.insert(t);
```

在上面的代码中我们在容器中创建了一个新的 Thing 对象，它具有和 t 一样的值（期间调

用了 Thing 的复制构造函数)。对于容器中的 Thing 对象进行的改动不会影响到 t，同样，对于 t 的改动也不会影响到容器中的 Thing。

- 当容器被摧毁时，它所包含的所有对象都会被摧毁。

有着引用语义的容器不会对值进行复制，它们只是在对象中存储着引用。(我们在此使用术语*引用*只是用来表示一个抽象概念；实际上它可以是 C++中的引用、指针或者其他的数据结构。)当容器具有引用语义时：

- 我们可以把一个对象“放到”多个容器中去，这时某个容器中只包含了指向该对象的引用。
- 将对象放置到容器中不会导致对象被复制。
- 当容器被摧毁时，它所包含的对象并不会被摧毁。
- 当“容器中”的对象被摧毁时，会出现一些问题。除非我们明确地提供了某种机制来告诉容器该对象被摧毁了，否则容器中对该对象的引用将会变成空悬引用。

8.1.3 容器和指针

现在我们来考虑一下对于前面两个问题的不同答案之间的组合：

有着值语义的异型容器

在 C++中使用这样的容器是很困难的一件事情。一个有着值语义的容器必须往容器中复制对象；但一个异型容器在复制对象时得不到它的确切类型，因而也无法调用到该对象的复制构造函数。

有着值语义的同型容器

我们可以用模板来直接实现这种类型的容器。

有着引用语义的容器（同型或异型）

当构建一个支持引用语义的容器时，对于同型和异型容器（或句柄）来说，我们都可以得到同样的行为。前面所说的句柄可以是 C++中的常规指针，或者是某种类型的智能指针。对指针的使用是可以被分享的。对一个异型容器来说，指针指向的是一个公共的基类，这使得我们可以向容器中插入任意指向该基类的派生类的指针。当这样的容器被摧毁时，被这些指针所指向的对象不应该被摧毁(因为它们有可能正被其他容器所共享，或者对象可能是一个不能被明确摧毁的自动/静态对象)。

对容器的设计者来说，首先应该考虑到的是支持值语义的同型容器。我们可以直接用模板来实现它们；另外，如果用户需要获得引用语义的行为，他们可以选择使用一个包含指针的容器。不管我们如何努力，不能被直接支持的容器（具有值语义的异型容器）还是很难被实现。

8.2 示例：Block

我们的第一个容器 Block 被用来提供和 C++中数组相似的抽象，而且我们还可以在程序运

行期间对它的长度进行改动。[1]我们把它实现为一个具有值语义的同型容器，并让它支持任意可以被复制的类型。在我们的设计目标中，运行时的效率是必须被首先考虑的：对 Block 中元素的访问应该和访问数组中的元素一样快。下面就是 Block 的接口：

```
template <class T>
class Block {
// 私用的细节部分忽略
public:
    Block(int n_elem = 0);
    ~Block();
    void size(int); // 设置元素的个数
    int size() const;   // 取得元素的个数
    T& operator[](int);
};
```

在构造函数中的整型参数指定了 Block 对象的初始大小。根据 T 的类型不同，容器中的某个元素的初始化过程也不会相同：

- 如果 T 是提供了构造函数的类，那么在初始化 T 时，T 的缺省构造函数会被调用；
- 如果 T 中提供了构造函数但没有缺省构造函数，那么模板就不能被实例化；
- 如果 T 中没有构造函数，那么它的初始值将是一个未定义的值。
- 上面代码中的两个 size 函数可以用来对 Block 中元素个数进行设置和读取。被重载的 operator[]操作符使得我们可以像使用 T 数组那样使用 Block<T>：

```
// 计算出费波拉切数列中的前 40 项：
Block<int> fib(40);
fib[0] = 1;
fib[1] = 1;
for (int i = 2; i < fib.size(); ++i)
    fib[i] = fib[i-1] + fib[i-2];
```

由于 operator[]返回的是一个非常量的引用，我们也就可以在赋值表达式的左边来调用 operator[]。

提供对 Block 大小的改动，我们也可以计算出斐波纳契数列中接下来的 40 项元素：

```
fib.size(80);
for (int i = 40; i < fib.size(); ++i)
    fib[i] = fib[i-1] + fib[i-2];
```

我们的 Block 模板会保证它里面所有的元素都被存储在一个连续的内存空间中。这使得我们可以像使用指向数组的指针那样使用指向 Block 的指针，也可以简化我们对于那些“使用指向常规的 C++数组的指针作为其参数”的函数的调用：

```
int sum_of_ints(const int*, int count);
```

1 这个类来自于 USL 标准组件库中的 Block 类；它的初始作者是 Andrew Koenig。

```
// ...
int sum = sum_of_ints(&fib[0], 40);
```

对于 operator[]调用的结果在下次调用 size(int)前都将会被保证为有效。用一个大于 size()的索引来调用 operator[]会得到一个未定义的值；由于运行时的效率是我们最关注的方面，我们也就没有在访问时加上会导致时间开销的检测机制。

8.2.1 实现 Block

Block 的实现很简单：我们在模板中维护了一个指向连续放置的 T 数组的指针以及一个计数器。出于效率考虑，Block 中除了用来改变大小的函数外的所有函数都被说明为内嵌函数：

```
template <class T>
class Block {
private:
    T*  elements;
    int n_elements;
public:
    Block(int n_elem = 0)
    : elements(new T[n_elem]), n_elements(n_elem) {}
    ~Block()    { delete [] elements; }
    void size(int);
    int size() const { return n_elements; }
    T& operator[](int index) { return elements[index]; }
};
```

在构造函数中，我们创建了一个 T 的数组，并将计数器初始化为它里面的元素个数；在析构函数中，我们会删除该数组。（注意：n_elem 可以为 0，这会导致我们创建一个有着 0 个元素的数组；在 C++中这样做是合法的。）我们在[]操作符中提供调用 C++中内建的[]操作符来提供对索引元素的访问。

为了重设置 Block 的大小，我们会为数组重新分配空间并将已有的对象复制到新的空间中去：

```
template <class T>
void
Block<T>::size(int new_sz) {
    if (new_sz != size()) {
        T* new_ele = new T[new_sz];
        if (new_ele == 0)
            fatal_error("Out of memory");
        int n_copy = (new_sz < size())
        ? new_sz : size();

        for (int i = 0; i < n_copy; ++i)
            new_ele[i] = elements[i];

        delete [] elements;
```

```
            elements = new_ele;
            n_elements = new_sz;
        }
    }
```

如果对 size(int)的调用会导致 Block 的长度增加，我们就需要将原有的所有对象都复制到新的存储空间中去。如果 Block 的长度变小，我们只需要复制那些新空间能够容纳的元素（也就是说，Block 被截断了。）

注意：在缩小 Block 的长度时，我们不需要对所有的元素进行复制。将一个 Block 长度缩小会导致那些被“截断”的对象被摧毁，但我们没有办法来有选择性地摧毁数组中的某些元素。[2]我们要做的就是为新的数组分配空间，将需要的元素复制到新的空间里面，然后再摧毁原有的那个数组。

8.3 有关 Block 的设计细节

现在我们的 Block 模板可以工作了。我们将来继续关注一些设计它时所采用的细节问题：

8.3.1 自动调整长度

我们首先来考虑这两个问题：为什么 operator[]在下标过大的情况下会产生未定义的行为？为什么我们不能在它里面检测数组的长度并在需要时调整数组的大小呢？

```
// 对于 Block 的一个危险的改进
template <class T>
T&
Block<T>::opertaor[](int indx) {
    if (indx >= size())
        size(indx+1);   // indx 以 0 为开始下标
    return elements[indx];
};
```

经过上面的改动后，当我们引用到一个超出数组边界的元素时，Block 的长度会自动增加；在这里面我们不需要明确地调用 size：

```
Block<int> block;
block[5] = x;
```

不过还是有着两个理由让我们不用这么做。在我们先前的实现中，对 Block 中的元素的访问速度和对数组中元素的访问速度是一样快的。而在修改后的版本中我们就无法继续保证这一点，这就会导致我们“尽可能地提高运行时效率”这个主要的设计目标遭到破坏。

这种改变带来的第二个问题是：对 operator[]的每次调用都会导致一次内存的重新分配。

2 即使我们可以通过明确地对对象调用析构函数来摧毁独立元素，在数组被摧毁时这些元素还会被再次删除，这将导致未定义的结果。

对于同一个表达式中出现了两次 operator[]的调用这种情况来说，这无疑是一个噩耗：

```
Block<int> ray;
// ...
ray[x] = ray[y];    // 不安全的做法
```

上面的表达式的执行顺序可能会是：

1. 求出 ray[x]的值，它是一个指向 ray 中第 x 个元素的指针

2. 然后我们再来求 ray[y]的值，此时我们会对 ray 的空间进行重新分配（operator[]）。所有 ray 中的元素都将被复制到新的空间中去，原有的元素将会被删除

3. 最后我们会进行赋值的操作，它会把 ray[y]所代表的内存中的值复制到 ray[x]所指向的内存中去（ray[x]所指向的内存并不在新的 ray 所指向的内存块中）。这就将导致程序产生未定义的结果。

下面是另外一个例子：

```
void walk_sequence(int* start, int* end);
walk_sequence(&ray[0], &ray[10]);
```

如果我们首先是对 ray[0]求值，那么接下来的对于 ray[10]求值会导致数组空间的重新分配，此时 walk_sequence 中的第一个参数将会指向一块已经被删除了的内存。由于在 C++中没有指定函数参数的求值顺序；这样做很容易导致问题的出现；一旦我们更换了编译器，我们已有的能够正常工作的代码可能就会因为这个 bug 而崩溃。

如果不存在从容器中取得指针或引用的代码，那么容器的实现者就将轻松多了；但实际上我们总是经常需要使用到这样的指针或引用。我们可能会用它们来避免复制容器中的元素，或者是由于程序中的逻辑（对象不能被复制）使得我们不得不使用它们。如果容器中提供了返回指向它里面元素的指针或引用的操作，那么我们就必须在容器的接口中指出容器的长度有多大。我们应该避免实现类似于上面的 operator[]那样“会自动改变容器大小，并且它的返回值只在对该操作的下一次调用前有效”的操作。即使我们在文档中详细地描述了这种情况，当表达式的求值顺序取决于具体编译器时，我们还是很可能得到一些奇怪的 bug。

在我们先前的 Block 类中并不存在着这种问题：operator[]的返回值在下一次调用 size(int)前都将有效。当然，这样做的代价就是用户要去关注 Block 什么时候需要增长。

8.3.2 常见场合下的性能

当我们构建容器时，我们应该考虑的是在常见场合下它的性能特性。最常见的两种情况就是对容器进行初始化以及遍历该容器。

我们先来看看用户是如何对 Block 进行存储的。假设我们用了一个 Block 来存储来自流的输入，而且我们不知道流中的下一部分内容会是怎么样的。很显然，碰到这种情况我们需要在必要时对 Block 进行增长：

```
extern int gets(String&);  // 读入一行到String中
main() {
    Block<String> input;
    String line;
    int line_no = 0;

    while(gets(line)) {
        input.size(line_no+1);
        input[line_no++] = line;
    }
}
```

上面的代码可以正常工作，但它运行得很慢。问题出在对 Block 的 resize 操作会导致所有的元素都被复制，这就导致对于 size(int)的调用的复杂度为 O(size())。由于上面的循环会在每次迭代中都进行 resize 操作，随着输入的增加，循环的运行时间会呈二次方的形式增长。

对于这种问题的一种解决办法就是新增一个 reserve 函数，用它来为某个长度下限而不是上限的 Block 进行空间的分配：

```
    while(gets(line)) {
    input.reserve(line_no+1);
    input[line_no++] = line;
}
```

如果在 input 中至少有 line_no+1 个元素，那么对于 reserve 的调用不会对 input 有任何改变。否则，reserve 就会为 input 分配至少（也可以多于）line_no+1 个元素的空间。至于 Block 一次会增加多少个元素这个事情则属于实现的细节部分。

由于现在 Block 一次可以增加多个元素，我们就可以对前面的循环进行重写，以得到一个对 Block 存储时的线形时间复杂度。下面是 reserve 的一种可能的实现：

```
template <class T>
void Block<T>::reserve(int sz) {
    int new_sz = (size() == 0)? 1 : size();
    while (new_sz() < sz)
        new_sz *= 2;
    size(new_sz);
}
```

如果我们不得不对 Block 进行增长，那么我们就让它增长为原来大小的两倍（如果 Block 的大小为 0 时则对它增加 1 个元素的空间）。这样对于 reserve 的大部分调用都不会导致内存的重新分配。我们现在需要对前面关于指针或引用的有效性保证进行修改：现在它们只是在下一次调用 size(int)或 reserve(int)才被保证有效。

如果我们在 Block 的填充中使用了 reserve，那么内存的重新分配次数就是 $O(\log_2 size())$，对于 T 的复制总次数就是：

$$n+\frac{n}{2}+\frac{n}{4}+\frac{n}{8}+\cdots+0$$

此处 n 是 Block 中能够容纳元素的最大个数；我们也可以把上面的表达式改写为：

$$n(1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\cdots+0)$$

它的结果肯定要小于 2n；这就意味这填充 Block 的时间和它的大小成一个线形关系。

我们同时还应该考虑对 Block 进行遍历的性能。由于对 operator[]的调用会扩展为一个简单的数组引用，对 Block 进行遍历的速度应该和对数组遍历的速度一样快。

8.3.3 常见的编码惯用法

我们希望那些对容器的最常见使用的代码能够尽可能地简洁。如果我们回顾一下在循环中使用 reserve 的代码，我们就会发现，每次我们调用 reserve(int)时都会引发一个 a + 1 的操作，以此来将以 0 开始的索引转化为由 1 开始的索引。我们该如何避免这个额外的加法操作呢？

有一个办法就是：改变 reserve 的定义，确保我们在调用 reserve(n)时 Block 中至少会有 n + 1 个元素。现在我们就可以将填充一个 Block 容器的代码改为：

```
while(gets(line)) {
    input.reserve(line_no);
    input[line_no++] = line;
}
```

我们在此对这个细节进行讨论的原因就是：用户忽略掉它并不会带来什么影响。如果用户使用了+ 1 那种做法，程序仍然能够正确地运行：只是这时的 Block 会比它应该具有的长度稍微大一些。

8.3.4 和内建类型之间的冲突

当在设计模板时同时也考虑函数重载，我们必须对下面这种情况保持警惕：某些模板参数可能会导致函数声明的重复出现。例如，我们可能会希望向 Block 中添加一些额外的构造函数，以便于用户用已有的一些元素来创建一些小的 Block 对象：

```
template <class T>
// 对 Block 的有着风险的改进
class Block {
public:
    Block(int n);          // 包含 n 个具有缺省值的元素

    Block(T);          // 仅含一个元素的 Block
    Block(T, T);           // 包含两个元素的 Block
    Block(T, T, T); // 包含三个元素的 Block
    // 其他细节忽略
```

```
};
```

这使得我们可以编写如下的代码：

```
Block<String> names("Jane", "Doe");
```

上面的代码会创建一个具有两个元素的 Block 对象。

看上去上面的代码工作得很好，不过一旦我们试图实例化 Block<int>，它就将有问题了；用户将会得到一个实例化错误，因为现在我们有了两个 Block<int>::Block(int)的函数声明！

我们可以通过为 Block<int>提供一个特别的实例化版本来避免上面的错误出现（见 8.8 节），但这样做的结果仍然会令人迷惑：

```
Block<int> b(4);
```

上面的代码到底是创建一个带有一个值为 4 的元素的 Block，还是一个具有 4 个未初始化的元素的 Block 呢？在这种情况下，我们最好的做法就是不要去提供这些额外的构造函数。

8.4 带有迭代器的容器：List

在 Block 类中，我们使用了整数作为索引来遍历整个容器。对于 Block 来说，这样做可以得到很好的效果：因为它只能在一端进行增长和缩减，这样就保证了我们通过同一个整数总是能够引用到容器中的同一个元素（前提是该元素还存在于容器中）。

但对于那些可以在不是容器两端的位置进行元素的增删的容器来说，就不能再用整数来对它进行遍历了——因为对于元素的增删会导致其后续元素的索引值发生改变。如果容器代表的不是一个元素的一维序列时，同样不能用整数来对它进行遍历；例如，用整数来遍历一棵树这种做法就很麻烦。对于这些容器来说，我们最好的做法就是为它们提供一个独立的迭代器。在本节中，我们将会设计一个模板类 List，以及和它对应的迭代器 Listiter。List 是一个具有值语义的同型容器，它包含的元素也都是可以被复制的。为了简单起见，我们只允许在 List 上进行前向的遍历，而不允许逆向遍历。

List的接口

List 类[3]几乎没有任何接口：

```
template <class T>
class List {
// 此处忽略细节
public:
    List();
    int length() const;
};
```

3 该类的设计借鉴了 USL 标准组件库中的 List 类，它的最初作者是 Jonathan Shopiro。

为了提供对单个的链表元素的访问，我们需要一个迭代器。于是我们定义了 Listiter 来作为概念上用以指向 List 中元素间的间隙——它同样也可以指向 List 中的第一个元素前以及最后那个元素后（见图 8.1）。

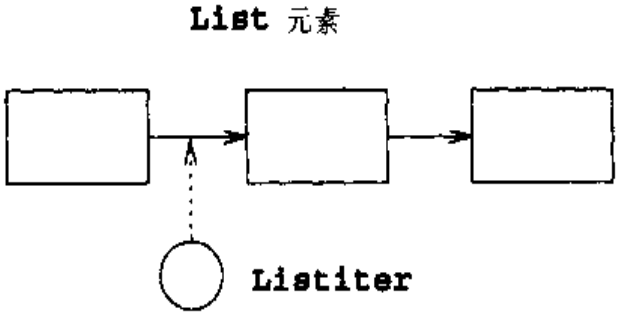


图 8.1 List 中的迭代器

Listiter 可以提供所有依赖于链表中位置的操作：

```
template <class T>
class Listiter {
// 此处忽略细节
public:
    Listiter(List<T>&);
    ~Listiter();

    void insert_after(const T&);
    void remove_after();

    int next(T&);
    int peek_next(T&) const;
};
```

我们在初始化 Listiter 时必须为它提供一个它所遍历的 List 对象的引用：

```
List<Thing> list;
// 与相关的 list 绑定...
Listiter<Thing> iter(list);
```

一旦被创建后，Listiter 对象就再也不能用来指向其他的 List 对象了。

我们将该功能放于一个独立的类（而不是 List）中的原因是：我们希望能够同时有多个迭代操作向同一个 List 中末端添加元素。那些会改变位置的迭代器不应该影响到其他的迭代器。（不过，不论是插入还是删除，我们都必须及时地在 List 中体现出来。）

迭代器：指向元素还是指向元素间的间隙？

让迭代器在概念上指向元素间的间隙有着一个重要的好处：当元素被插入和删除时，临近的迭代器行为就将变得很直观了。如果迭代器在概念上来说是直接指向一个对象，那么当该对象被删除时，对于迭代器来说会有什么情况发生呢——它指向的到底是原来元素前面还是后面

的那个对象呢？如果迭代器指向的是两个对象间的间隙，我们就可以避免这个问题的出现了。

从另一方面来说，在这种模型中，任何会影响到已有对象的迭代器操作都必须指明该操作会影响到的是该迭代器前的对象还是后的对象。直接指向对象的迭代器就不存在这个问题（虽说它们的“插入”操作也必须指明是在对象前插入还是在对象后插入）。

当设计一族容器时，我们必须优先考虑所有模型间的一致性，而不是单个模型间的特性；不管我们选择了什么样的模型，我们都必须在整个容器族中坚持使用它。

其他的迭代器操作

next 函数可以被用来遍历 List。如果迭代器没有处在链表的末端，那么对于 next 的调用就会将迭代器后面的那个元素复制到 next 的参数 t 所引用到的位置，然后迭代器会向前步进使得它指向参数后面的那个元素，并且函数返回 1。否则，next 的返回值将会为 0，并忽略其参数的作用。这样我们就可以这么写：

```
Listiter<Thing> iter(/*...*/);
Thing t;
while(iter.next(t)) {
    // 使用 t
}
```

peek_next 函数所提供的语义和 next 一样，不过它不会对迭代器进行步进。insert_after 函数会将它的参数插入到链表中去，并且在插入完成后迭代器指向的是新元素的前面。remove_after 函数会从链表中删除迭代器参数后面的那个元素；如果迭代器位于链表的末端，那么它会向外产生一个错误信息。

现在我们就得到了一个 List 的最小接口；真正用于产品中的 List 类应该具有比这个接口更多的功能（参见问题 9）。

实现List

List 对象中包含有一个指向 List_link 链表的头指针，在 List_link 中的每个元素都包含有一个指向链表中下一元素的指针以及我们真正需要的被包含的对象（参见图 8.2）。

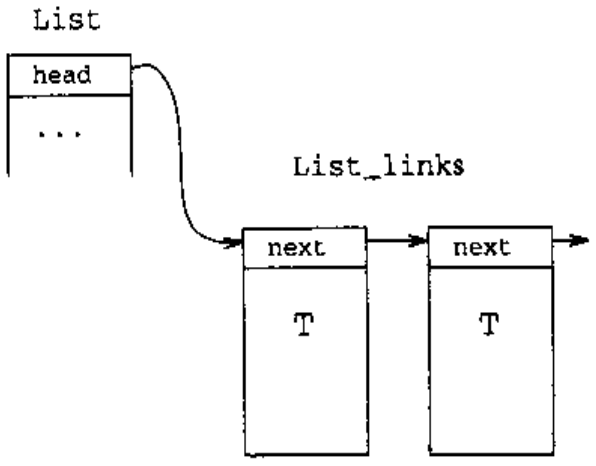


图 8.2 List 的实现

我们先来设计 List_link 类：由于它是一个实现类（相对于接口类来说），我们把它的所有成员都设计为私用的，并且让 List 成为其友类。List_link<T>中有两个数据成员：一个用来指向链表中下一元素的指针，以及链表元素 T。

```
template <class T> class List;
template <class T> class Listiter;
template <class T>
class List_link {
    friend class List<T>;
    friend class Listiter<T>;
private:
    List_link<T>*   next;
    T               data;
    List_link(const T&, List_link<T>*);
};
```

List_link 的构造函数的功能很简单，只是简单地对其数据成员进行初始化：

```
template <class T>
List_link<T>::List_link(const T& val, List_link<T>* nxt)
: data(val), next(nxt) {
}
```

接着，我们就开始设计 List 类了，我们让每个 List 对象中都包含有一个指向 List_link 链表中的头指针以及一个用来表示 List 中元素个数的计数器。(我们将在涉及到 Listiter 类时再来谈论成员 iters 的功能)：

```
template <class T>
class List {
    friend class List_link<T>;
    friend class Listiter<T>;
private:
    List_link<T>*       head;
    Set<Listiter<T>*>   iters;
    int                 n_elem;
public:
                List();
    int         length() const { return n_elem; }
};
```

List 的构造函数会创建一个空的链表：

```
template <class T>
List<T>::List()
: head(0),
  n_elem(0) {
}
```

然后，我们就开始实现迭代器了。虽然我们的抽象模型中将 Listiter 指向了两个对象之间的间隙，我们还是将它实现为指向一个对象的指针（因为 C++中不允许我们将指针指向两个对象之间的间隙）。我们假设指针指向的是两个对象中前面的那个对象；如果迭代器位于链表的开始处时，该指针将为一个空指针。

如果 Listiter 指向的是一个对象的话，那么当我们从 List 中删除一个对象时会发生什么现象呢？我们会将那个指针指向原来对象前那个对象（如果被删除的对象处于链表开始处时，我们就将该指针置为空）。由于允许同时存在多个指针指向被删除的元素，我们必须在 remove_after 中对所有和 List 相关的 Listiter 进行检测，并在需要时对它们的值进行调整。为了做到这一点，每个 List 中都必须维护一个由当前指向它的所有 Listiter 迭代器所组成的集合。

每个 Listiter 中都包含由一个指向它所迭代的 List 的指针以及一个指向其“前面”对象的指针（记住：在我们的抽象模型中，迭代器是指向两个对象间的，这个“前面”的对象就是这两个对象中位于迭代器前面的那个对象）：

```
template <class T>
class Listiter {
private:
    List_link<T>*   prev;
    List<T>*    the_list;
    List_link<T>** next_link();
public:
    Listiter(List<T>&);
    Listiter(const Listiter<T>&);
    ~Listiter();

    void insert_after(const T&);
    void remove_after();

    int next(T&);
    int peek_next(T&) const;
};
```

Listiter 的构造函数和析构函数必须更新 List 中的那个由迭代器所组成的集合：

```
template <class T>
Listiter<T>::Listiter(List<T>& list)
: prev(0),
  the_list(&list) {
    the_list->iters.insert(this);
}

template <class T>
Listiter<T>::Listiter(const Listiter<T>& iter)
: prev(iter.prev),
  the_list(iter.the_list) {
    the_list->iters.insert(this);
```

```
}

template <class T>
Listiter<T>::~Listiter() {
    the_list->iters.remove(this);
}
```

当其他的 Listiter 函数被实现后，（私用的）next 函数就很容易被得到；它会返回指向当前迭代器所指的后面那个 List_link 的指针：

```
template <class T>
List_link<T>**
List_link<T>::next_link() {
    if (prev)
        return &(prev->next);
    else
        return &(the_list->head);
}
```

insert_after 会创建一个新的 List_link，将它插入到链表中去，然后对 List 对象中的计数器进行更新：

```
template <class T>
void List_link<T>::insert_after(const T& ele) {
    // 插入新的 List_link

    List_link<T>** nextlink = next_link();
    *nextlink = new List_linke<T>(ele, *nextlink);

    // 更新 List 对象中的计数器

    the_list->n_elem++;
}
```

remove_after 会从链表中删除一个 List_link 元素，并对计数器进行更新；但我们还必须同时通知所有已有的迭代器“List 被改动了”这个事实：

```
template <class T>
void
List_link<T>::remove_after() {
    // 找到要被删除的 List_link
    List_link<t>* removed;
    List_link<T>** nextlink = next_link();
    removed = nextlink;
    if (!removed)
        internal_error("Remove from end of list");

    // 移除已有的 List_link
```

```
    *nextlink = removed->next;

    /* 找到所有指向被“删除”对象后面的迭代器，让它们指向原有对象“前面”
那个对象；如何使用USL标准组件库中的Set可以参见附注* */

    Setiter<Listiter<T>*> listiters(the_list->iters);
    Listiter<T>* iter;
    while(listiters.next(iter)) {
        if (iter->prev == removed)
            iter->prev = prev;
    }

    // 删除被移除的List_link
    delete removed;

    // 更新List对象中的计数器
    the_list->n_elem--;
}
```

peek_next 函数会检测下一个元素，但它不会改变迭代器的状态：

```
template <class T>
int
Listiter<T>::peek_next(T& ele) const {
    const List_link<T>** nextlink =
        ((Listiter<T>*)this)->next_link();
    if (*nextlink == 0) return 0;
    ele = (*nextlink)->data;
    return 1;
}
```

我们在 next 中调用 peek_next 来获取下一个元素的值，然后对迭代器进行步进并返回：

```
template <class T>
int Listiter<T>::next(T& ele) const {
    if (peek_next(ele)) {
        prev = *(next_link());
        return 1;
    }
    return 0;
}
```

正确地编写这些函数会需要 List 的实现者一定的精力；但一旦我们得到了正确的函数后，

* 如果我们使用了 USL 标准组件库中的 Set 类，我们可以使用 Setiter 来返回一个指向 Set 的迭代器，这样我们就可以得到 remove_after 中的循环如下：

```
Listiter<T>** piter;
while(listiters.next(piter)) {
if ((*piter)->prev == removed)
   (*piter)->prev = prev;
}
```

用户就不再需要去关心它们的正确与否了。

8.5 迭代器的设计细节

我们先来看看迭代器和它所迭代的容器之间的一些交互。

8.5.1 当容器被改变或摧毁时，请更新迭代器的值

当容器的内容被改变时（如，向容器中插入/删除元素），当前所有指向它的迭代器都必须知道这种改变。我们的 List 类会在它的 iters 成员中记录所有当前与它相关的迭代器。当迭代器对 List 有所改动时，它会去检测所有和 List 相关的迭代器，看看它们是否会受到这次改动的影响。当容器被摧毁时可能也会有迭代器的存在。在随后对于这些迭代器的引用中，我们应该给出一个明智的做法，如：向外抛出一个异常或者假设迭代器指向的是一个空的容器。

下面就是另外一个可选方案，让容器中的迭代器在概念上指向元素本身，而不是元素之间：如果容器中被删除的元素被多个迭代器所指向，该元素在所有这些指向它的迭代器被摧毁之前是不会被摧毁的。迭代器中维护了对该元素的使用次数的计数器。这样做的缺点在于：每次我们对迭代器进行步进时，它必须去调整对象中的那个计数器。

8.5.2 next()的设计

使用迭代器来遍历容器是一个很常见的操作，所以我们必须尽可能简洁（输入最少）及高效地实现 next 函数。由于 next 在概念上会返回两个值，用于指示操作是否成功的标志以及成功时得到的下一个对象，这就使 next 变得复杂起来。实现这种做法的一种选择是让它返回一个 Pair<int, T>；当如果我们真的这么设计的话，使用 next 函数的迭代就可能会是如下情况：

```
Listiter<Thing> iter;
Pair<int, Thing>    rc;
while(rc = iter.next(), rc.left()) {
    // ...
```

在这种情况下，它并不如用一个引用参数来存储返回的 T 对象来得简洁：

```
Listiter<Thing> iter;
Thing   val;
while(iter.next(val)) {
    // ...
```

对大多数类型来说，第二种做法就已经足够了。然而，在这种做法中，对于每次的迭代，用户都必须先创建一个空的 T 对象，然后用 next 来对它进行赋值。当 T 中没有提供缺省构造函数时，这样做可能会有问题。

我们希望我们的模板可以和尽可能多的类一同工作。所以，为了让用户使用我们的模板去

处理那些没有提供缺省构造函数的类，我们可以提供另一个版本的 next。如果编译器支持指向容器内部的指针，那么这个新版本的 next 就可以“返回”一个指向下一元素的指针：

```
template <class T>
class Listiter {
public:
    // ...
    int next(T&);
    int next(T*&);
};
```

显然，由 next(T*&)所“返回”的指针的有效生命周期必须被用户明确指定。由于我们不会去移动 List_link 链表中的对象，我们就可以保证当元素处于 List 中时，指向它的指针都将会有效。

如果编译器不支持指向容器内部的指针，那么我们最好将 test 和 next 这两个函数分割开来：

```
template <class T>
class Listiter {
public:
    // ...
    int next(T&);
    T   next();
    int at_end() const;
};
```

next()函数将返回一个 T，但它不能让我们正确处理 List 末端这种情况。为了解决这个问题，我们新增了一个 at_end 函数，当迭代器指向链表最末端元素后时，它将会返回 1。当 at_end() 返回 true 时，next()（而不是 next(T&)）函数就会向外报错（通过抛出异常的方式——前提是我们正在使用一个支持异常的 C++编译器）。

使用这两个函数的代码大致如下：

```
Listiter<Thing> iter(/* ... */);
while(!iter.at_end()) {
    Thing t(iter.next());
    // ...
}
```

虽然上面的代码不如最初的那么简洁，但它里面没有包含空的 Thing 对象的创建，也没有涉及一个返回指向 List 内部指针的函数。

8.5.3 常量迭代器

在 Listiter 的构造函数中，我们使用了 List<T>&而不是 const List<T>&。这是因为我们的

迭代器可以通过调用 insert_after 或者 remove_after 来改变 List 的状态。

我们可能会希望提供一个 Cosnt_listiter 用以对 const List 继续遍历。Const_listiter 的构造函数中会使用一个 const List<T>&的参数——但它不能包含有 insert_after 和 replace_after 这样的会改变 List 状态的成员函数。一旦有了这么一个 const List 对象，我们就可以声明接受 const List&作为参数的函数；在这些函数中，我们就可以使用 Const_listiter 来检测、而不是改变这个 const List&参数：

```
void examine(const List<int>& list) {
    Const_listiter<int> iter(list);
    int i;
    while (iter.next(i))
        // 等等...
```

上面代码中 examine 的调用方就知道：在函数中 List 参数的状态不会被修改。

我们可以在 Listiter 中做到 Const_listiter 所能够做到的所有事情，甚至比它还要多；这样的话，我们就会考虑将 Listiter 派生自 Const_listiter。（也就是说，Listiter 是一个可以改变 List 状态的 Const_listiter。）我们将查询功能的函数放到 Const_listiter 中去实现，而插入和删除这样的函数则留给了 Listiter：

```
template <class T>
class Const_listiter {
private:
    List_link<T>*   prev;
    List<T>*    the_list;
public:
    Const_listiter(const List<T>&);
    virtual ~Const_listiter();

    int next(T&);
    int peek_next(T&) const;
};

template <class T>
class Listiter : public Const_listiter<T> {
public:
    Listiter(List<T>&);
    ~Listiter();

    void insert_after(const T&);
    void remove_after();
};
```

我们选择使用了公有继承（因为我们继承了接口）；这就使得我们可以将一个 Listier&作为参数传递给需要接受 Const_listier&参数的函数。

将一个 Listiter 和 const List&绑定到一起的尝试注定是要失败的（因为 Listiter 的构造函数需要的是一个非常量的引用）。通过 Const_listiter 来改变一个 List 对象的状态的做法也无法成功，因为在 Const_listiter 类中并没有声明这样的函数。当然，我们可以通过将一个常量引用强制转换为一个非常量引用来绕开这个机制；但通常情况下使用常量引用都是个正确的选择。

所有的特性都有其代价：如开发时间，代码尺寸和性能，代码的易理解性等。这也同样适用于我们对常量迭代器的评价。对于一个小的、使用范围也不大的模板来说，使用常量迭代器带来的好处还不足以弥补使用它带来的复杂度。对于一个大范围使用的模板（如 List）来说，使用它可能就会带来一些便利。和大多数 C++中的有趣问题一样，这完全是一个评价尺度的问题。

8.6 性能上的考虑

使用模板可以使得程序员避免设计大量的功能类似的新类，但对于编译器来说则没有得到任何的好处。对于大程序来说，编译器在实例化模板时所耗费的时间相当可观。我们将在这一节中讲述一些可以减轻编译器负担的模板技术。

8.6.1 将公共逻辑放到一起

我们期望某些通用的容器（如 List）可以在一个程序中被多次实例化。当设计这样的模板时，我们应该尽量将那些和参数类型无关的代码放到一个单独的地方，这样的话，该模板的多份实例就可以共享惟一的一份这样的代码。这样做可以避免编译器在每次实例化时都编译同样一份代码，程序的大小也会因为少掉的多份同样代码而得到精简。

找到这种代码的最好机会就是那些处理不清楚自己所指向的对象类型的指针的操作。我们通常都可以使用 void*来实现这些操作。所有处理 void*的代码都被共享，它们的返回值会被模板类中其他的（非共享）逻辑转换为真正适合它们的类型。

例如，每个 List<T>中都包含有一个 Set<Listiter<T>*>，编译器也会因此不得不为每个 List 实例实例化出不同的 Set 类。为了避免这种情况的出现，我们对 List 进行了修改，现在它维护的是一个 Set<void*>，而不是 Set<Listiter<T>*>：

```
template <class T>
class List {
    friend class List_link<T>;
    friend class Listiter<T>;
private:
    List_link<T>*   head;
    Set<void*>      iters;
public:
                    List();
    int             length() const;
```

```
};
```

所有从 Set 中返回指针的操作现在都必须将返回值转换为 Listiter<T>[4]：

```
template <class T>
void
Listiter<T>::remove_after {
    // ...
    Setiter<void*> listiters(the_list->iters);
    void* iter;
    while (listiters.next(iter)) {
        if (((Listiter<T>*)iter)->prev == removed)
            ((Listiter<T>*)iter)->prev = prev;
    // ...
}
```

上述的修改使得我们的代码要比以前晦涩了一些，但它带来的好处就是：所有在同一程序中的 List 现在都可以共享一份 Set<void*>的实例了。

当然，我们不能将这种改动推广到所有的容器上去。通过使用 void*，我们的实现就丧失了编译器会为我们所做的一些静态类型继承功能，并且程序也变得比较难以理解。我们必须在这种不便和它带来的收获——这依赖于 List 在一个典型程序中可能出现的实例化次数和每次对 Set 进行实例化带来的额外开销——之间取得一个平衡。如果这两者所带来的影响都很小的话，我们最好就不要把公有逻辑收集到一处。

如果希望能够进行更多的接合，我们也可以让所有的模板类都派生自一个公共的（非模板）基类，并在基类中实现那些和 T 无关的逻辑：

```
// 有关链表的公共逻辑：
class List_link_base {
private:
    List_link_base* next;
    // ...
};

// 和 T 相关的逻辑：
template <class T>
class List_link : public List_link_base {
private:
    T data;
    // ...
};
```

我们在上面的代码中把 next 指针放到了 List_link_base 类中去了，只有那些和 T 相关的代码才被放置到 List_link 模板中。我们也可以以同样的方式来对 List 和 Listiter 进行划分。

4 和 ANSI C 不一样的是，在 C++中不存在着从 void*到其他指针间的隐式转换。要想将一个 void*转换为其他类型的指针，我们必须提供一个明确的转换操作。

只有在模板展开所占的空间成为一个严重的问题时，我们才应该这么做；将一个容器划分为类型无关和类型相关的两部分会使得事情变得非常复杂。

8.6.2 内嵌化所有模板成员

使用模板的一个主要开销就在于它的实例化时间；如果模板的成员函数是内嵌的话，那么它的实例化时间就将变得不再重要了。我们仍然应该避免将大的函数内嵌化——由此所带来的额外的空间开销和编译时间甚至会超出因为模板实例化的减少所带来的好处。然而，在设计模板时，我们应尽量使所有成员函数内嵌化。

8.7 对模板参数的限制

当模板中的逻辑和其模板参数在很大程度上都无关时，模板的威力就可以得到充分的展示。例如，用来管理一个由 int 组成的栈和管理一个由 char*组成的栈的逻辑是一样的；我们就可以只编写一个模板来处理这个逻辑。然而，还是有很多的模板都会对其类型参数有着一些特定的要求的。

例如，假设我们有一个 List::max()函数，我们用它来找出链表中最大的那个元素，它里面使用了 operator<来对两个元素进行比较：

```
template <class T>
T List<T>::max() {
    Listiter<T> iter(*this);

    T biggest;

    // 将“最大”的那个元素初始化为第一个元素的值

    if (iter.next(biggest) == 0)
        internal_error("max of empty list");

    // 遍历其他的元素，如果碰到比当前“最大”的元素还要大的元素，把它记录下来

    T ele;
    while (iter.next(ele))
        if (biggest < ele)
            biggest = ele;

    return biggest;
}
```

只有当 T 有着缺省构造函数，两个 T 可以用 operator<来进行比较并且 T 之间可以赋值时，上面的函数才可以通过编译。对于那些没有支持前面所有限制条件的 T 来说，“从它们组成的

List 中去寻找最大的那个 T”这个尝试将会导致一个实例化时的错误出现。

我们应该把对类型 T 的要求精确地提出来。我们并不需要一个接受两个 T 参数的 operator< 函数；只要下面的代码片段：

```
if (biggest < ele)
```

可以被顺利编译，我们的 max 就可以正常工作了。例如：

```
class Number_string {
    // 此处忽略细节
public:
    Number_string(const char*);
    operator int() const;
};
```

即使我们没有提供 operator<(Number_string, Number_string)，我们仍然可以调用 List<Number_string>::max。这是因为将<操作符应用到两个 Number_string 上时会导致这两个 Number_string 进行隐式转换得到两个 int，然后我们再对这两个 int 进行比较。

在某些情况下，模板和宏是一样的：在被实例化前，我们并不能了解到模板的真正含义。尤其是，面对一个给定的实例，在对它进行实例化之前，我们并不清楚它能否正确地通过编译，如果能够顺利编译的话，它是否会需要其他模板的参与。因此，List 类的提供者必须将 max 函数的需求给详细记录下来。

8.7.1 不同的成员函数可能有不同的限制

按照 C++中的规则来说，在模板中有一个或多个成员函数都还没有被实例化时，我们也可以使用该模板——前提是那些没有被实例化的函数没有被程序所使用。这意味着“List<T> 中的 T 必须用<来进行比较”这个说法将不再正确；确切地说是 List<T>::max 要求我们必须使用<来对 T 进行比较。如果不去调用 List 中的 max，我们也可以使用不能用<来进行比较的对象所组成的 List。

对于这个规则有两点要注意：首先，所有在模板类中声明的虚函数都被视为“被调用”；如果我们使用了某个模板类，那么它里面的所有虚函数都必须被正确地实例化。其次，某些编译器实现会实例化模板类中所有的成员函数，而不仅仅是那些被使用的成员；如果我们正在用的就是这样的编译器，对于那些不能被正确编译的成员，无论是否使用了它们，我们都会得到一个实例化错误信息。

用于展示更多限制的示例

某些模板会强制要求它的类型参数是一个自定义类型：

```
template <class T>
class Persistent : public T {
// 此处忽略细节
};
```

除非提供一个特化版本（见 8.8 节），我们不可能得到一个 persistent <int>：因为它不可能成为其他类的基类。下面是另外的一个示例：

```
template <class T>
class Smart_pointer {
public:
    T operator->() const;
};
```

在此，T 必须是一个指向自定义类型的指针，或者是一个声明了 operator->的对象。

如同我们所见到的那样，模板类的接口要比常规类复杂得多。除了抽象模型和可用的成员函数列表之外，模板类的接口还包括它对于类型参数的各种限制。

8.8 模板特化

模板可以用来为编译器提供一个由类（或函数）组成的无限集合；但我们也可以提供定义一个特化版本来为单独的实例提供支持。

8.8.1 为性能所做的特化

模板特化常用于为某个特殊的类型参数集合提供备选的实现。对于我们的 List 类来说，缺省的 List<char>已经可以正常工作了，但我们也可以通过将字符串存储在一块连续内存中而得到一个更快的版本。对于这个版本的 List<char>来说，插入和删除操作将只是对链表中的元素进行一些复制操作，对于 char 来说，这样做的开销要小多了：

```
template <class T>
class List { /* 同上 */ };

class List<char> {
    friend class Listiter<char>;
private:
    char*   data;
    int len;
public:
    List();
    int length() const { return len; }
};
```

特化版本中的非内嵌的成员函数（以及静态数据成员）必须在每个程序中都被定义一次，这一点和其他的类是一样的：

```
// 文件 List_special.c
#include <List.h>
```

```
List<char>::List()
: data(0), len(0)
{}
```

我们同样应该对 Listiter<char>进行特化，以使它可以和特化版本的 List<char>一同工作。

对于特化版本来说，我们并不要求它去查询通用版本的模板类中具有相同名字的部分。而且，特化版本应该提供和通用版本一样的公用接口。如果两者间的接口不一致，我们还有什么理由把这个类叫做 List 呢？

8.8.2 用于绕开模板限制的特化

当某些特殊的类型参数会破坏通用模板类的限制时，我们也可以使用特化。例如：

```
template <class T>
class Persistent : public T {
// ...
};

class Persistent<int> {
// ...
};

class Persistent<char> {
// ...
};

// 等等 ...
```

在此，缺省的 Persistent 模板要求它的类型参数必须是一个自定义的类型。我们通过提供特化版本将它扩展为也可以处理内建类型。（当然，这样做并没有完全解决问题，因为内建类型的种类也是无限多的；Persistent 的接口必须允许用户在必要时能够编写自己的模板特化。）

有时我们也可能仅仅为不能正确编译的成员函数提供特化。编译器应该在链接时找到这些特化了的成员函数，并避免为它们产生其他实例。

8.8.3 特化模板函数

和模板类不同的是，在调用模板函数时，它的模板参数并不需要被明确指出；我们可以通过实际参数来推导出这个类型参数：

```
template <class T>
void swap(T& left, T& right) {
// 此处忽略细节
}

String s, t;
```

```
swap(s, t); // 调用了模板函数 swap(String&, String&)
```

通过定义一个和模板相匹配的函数，我们就可以为模板函数提供一个特化。在本例中，所有名字为 swap，并接受两个引用到同样类型的参数的函数都是我们 swap 模板的一个特化[5]：

```
void swap(String& l, String& r) {
// Specialization code to swap Strings
}
```

在上面的代码中，虽说我们的 swap 模板声明在该函数的定义范围内并不可见，我们的 swap(String&, String&)函数还是被编译器视为 swap 模板的一个特化版本！按照现有的语言规则来说，将程序中和模板函数相匹配的函数定义视为该模板的特化这件事情是由编译器完成的。这就意味着我们在调用模板函数时，如果正好能够匹配到库中的另外函数的定义，我们将不会得到链接错误；在库中的那个函数会被自动视为模板函数的特化。

让我们考虑一下如下的操作系统的应用：

```
// 建立两个交换用的文件系统分区：主分区和备份分区
void swap(String& primary, String& backup) {
    // 有关文件系统的复杂逻辑
}
```

如果我们将一个目标文件（它里面调用了 swap 模板）和一个包含由定义该函数的库一同链接时，我们调用到的 swap 将会是库中的那个函数，这就将导致程序得到一个奇怪的结果。

要避免上述情况，我们最好不要大量使用模板函数，并且不去用那些常见的名字（如 read 和 write）来作为我们模板函数的名字。

8.8.4 特化和内嵌函数

C++语言规则要求：如果使用了模板特化，它就必须在程序的某处被定义；这意味着编译器只有在链接时才可以对模板进行实例化——因为我们可能会从库中找到某个特化版本。不过对于这样做还是有一个异常情况：内嵌函数可以在编译期间被扩展，即使当函数的特化只有在链接时才能被确定也是如此。对于这种异常情况的最安全做法就是：在所有对该类的成员函数进行的调用之前就声明类的特化版本：

```
template <class T>
class List {
public:
    int length() const { return len; }
    //此处忽略细节...
};

class List<char> {
```

5 译注：根据现在的 ISO/ANSI C++来说，这样的说法是不正确的。确切地说，具有同名的函数只能是模板函数的一个重载函数，会被一同列入到函数重载的解析过程中去。

```
public:
    int length() const; // 由于某种原因，它是非内嵌的
    //此处忽略细节...
};

void walk(List<char>& list) {
    int i = list.length();// 调用了（非内嵌的）List<char>::length()
```

8.8.5 用户提供的特化

模板的设计者可能会要求某个模板的用户都编写自己的模板特化来提供某些和模板参数类型相关的功能。例如，Set<T>模板可能需要一个 T 上的散列函数。Set 的性能在很大程度上都依赖于该散列函数的性能和质量；不过我们找不到一个可以适用于所有类型的好的散列函数。对 void*来说工作得很好的算法不一定能在 String 上面也工作得很好。所以某个使用 Set 类的用户都将不得不为他们的 T 来编写一个合适的散列函数。

对于 Set 的一种可行的做法就是将该散列函数作为一个静态的成员函数声明到模板中去，但我们并不为它提供定义：

```
    template <class T>
class Set {
public:
    //此处忽略细节
    static unsigned hash(const T&);
};
```

如果没有人提供一个该散列函数的特化版本的话，使用 Set 的用户代码：

```
#include <String.h>
unsigned Set<String>::hash(const String& f) {
    // ...
}
```

将会得到如下的链接错误：

```
Undefined Set<String>::hash(const String&)
```

Set 的提供者也可以提供一个缺省的散列函数；但由于我们现在对 T 还一无所知，我们目前所能够利用到的也只是该散列函数参数的地址：

```
template <class T>
unsigned Set<T>::hash(const T& t) {
    return (unsigned)(&t);
}
```

这样即使在用户没有提供散列函数时，他们仍然可以使用我们的 Set 类；不过此时的效率就可能不尽如人意——毕竟对于特殊类型的 T 来说使用地址并不是一个好的散列函数。这又

是一个评判尺度的问题：直接使用 Set 的便利性（将散列函数的设计延后到了需要时再进行）可能会让用户忘了去编写自己的散列函数以获得更好的性能。

使用那些需要用户提供模板特化的类的用户必须了解如何编写这个特化以及牢记在每次使用该模板时都应该做什么。我们同样还可以使用其他方法来消除这种负担。

作为构造函数的参数

我们可能会让用户把散列函数作为一个参数传递给类的构造函数：

```
template <class T>
class Set {
public:
    Set(unsigned (*HASH)(const T&));
    // ...
};

// ...

extern unsigned hash(const Thing&);

Set<Thing> tset(hash);
```

这样做会有一些缺点。无论何时 Set 被创建，用户都必须指定一个散列函数；每次对该散列函数的调用都是通过指针来进行的，这也就意味着我们不能把它内嵌化。

作为表达式参数

我们可能会决定将该散列函数作为模板的一个表达式参数：

```
template <class T, unsigned (*HASH)(const T&)>
class Set {
public:
    Set();
    // ...
};

extern unsigned hash(const Thing&);
Set<Thing, hash> test;
```

这样散列函数就可以是一个内嵌函数了，但现在我们将不得不在每次提及 Set 类时都指定出散列函数来！实际上，用户可能会因此被迫发明出一个新名字并用 typedef 来将 Set 定义到该新名字上去：

```
typedef Set<Thing, hash> Thing_set;
```

这样做不但会迫使用户发明新的名字——这会使得代码变得不那么直观，尤其是用户没有一个统一的命名规范时更是如此。很显然，Set<Foo>和 Set<Bar>看上去很相似（但还是不同）；但对于 Thing_set 和 Set_of_Things 来说呢？它们是一样的，还是两个不同的类呢？

纯虚函数

我们还有着另外一个选择：我们可以在模板中声明（但不是定义）一个纯虚函数来提供我们需要的功能：

```
template <class T>
class Set {
public:
    Set();
    virtual unsigned hash(const T&) = 0;
    // ...
};
```

这样，用户就必须从 Set<T>派生出新的类并为它提供合适的散列函数：

```
class Thing_set : public Set<Thing> {
public:
    unsigned hash(const Thing&);
};

unsigned Thing_set::hash(const Thing& t) {
// ...
}
```

这就允许用户在同一个程序中拥有多个 Things 的集合，它们之间的不同也仅仅是体现在散列函数的不同之上。从另一方面来说，对于用户的负担变得更重了，他们必须为某个新的 Set 类发明出一个新名字来；而且每次对于散列函数的调用到现在也都处理一次虚函数的调用。

8.8.6 最佳方法

上面几种方法的主要优点在于：它们使得程序中可以包含多个使用了不同散列函数的 Thing 对象的集合。在我们的 Set 类中，我们并没有采用这些解决方案：因为我们期望的是对于一个给定的类型 T，程序中所有的 Set<T>都使用同一个散列函数。（如果做不到这点，我们就不能使用用户提供的模板特化。）用户提供的模板特化允许用户只在程序中对散列函数进行一次性的指定，然后就可以对它不闻不问了。

假设有一个特殊的模板，它需要外界用户提供的功能，但这种功能又会随着对象的不同而不同？如果我们希望每个对象都有其独特的功能，那么向构造函数中传递函数指针看上去就应该是我们的最佳选择了——因为其他的方案都会导致新类的产生。如果被提供的功能是一个小的、很关注性能的、我们期望将其内嵌化的函数，那么我们就可以把它作为一个表达式参数传递给模板。除此之外，从一个公共的基类派生这种方式相对来说就最简单，也最容易被理解（能够理解虚函数的 C++程序员要多过能够理解模板的表达式参数的程序员）。

8.9 小结

- 用模板实现的容器应该是同型的。我们可以提供使用一个包含有指向公共基类的指针（或“智能指针”）的容器来获得一个异型容器的效果。
- 指向容器内部的指针或引用的有效生命周期应该由容器的提供者来指定。理想情况下的生命周期应该是“直至容器被摧毁为止”；有着“直到下一次调用该操作”这样的生命周期应该被避免。
- 迭代器应该指向元素之间或元素本身。在这一点上，由容器组成的集合应该达到共识。
- 模板中的每个成员都可以对其模板参数加以限制
- 将公用的逻辑独立出来并被所有实例共享可以使得常用模板性从中获益。
- 模板可以被特化。特化可以一直推迟到链接时才被确定。

8.10 问题

1. C++中使用了 new 来代替 malloc，delete 来代替 free。那么对于 realloc 来说，它的替代者是谁呢？在 C++使用 C 中的函数是否安全？如果不安全的化，你能不能用一个类来提供同样的功能呢？

2. 我们的 Block 类允许我们这么写：

```
&blk[x]
```

但不允许这么写：

```
blk+x
```

我们该如何对它进行改进，以使得用户可以按照下面的那种方式书写表达式并获得和前面一样的效果呢？（提示，这样写

```
blk
```

应该是一个合法的表达式。）

3. 设计出 Block 的一个替换方案，它的首要设计目的是使用时的安全性，而不是运行时的性能。我们应该如何修改我们的设计？这样的改变对于它的性能来说会有什么样的影响呢？

4. 设计出 Block 的一个替换方案，它可以在安全的形式下自动进行大小调整，使得：

```
blk[x] = blk[y];
```

即使在上面的某个引用会导致容器进行重新分配时也是安全的。为了做到这点，我们应该牺牲最初的 Block 类中的哪个（些）特性呢？

5. 下面的 Pair 模板有什么问题？

```
template <class Left, class Right>
```

```
class Pair {
//此处忽略细节
public:
    Pair(const Left l, const Right& r);
    Left left();
    void set(const Left&);
    Right right();
    void set(const Right&);
};
```

6. 在什么情况下，通过使用 reserve 而不是每次循环都调用 size 向 Block 中增加大小会变慢？

7. 如果 Block::reserve 决定要对 Block 进行增长，它采取的方式是增长为原来的两倍。我们还可以使用其他的增长策略吗？（提示 1：请考虑对小的块进行填充的性能。提示 2：请考虑在对大的块进行填充时采用 malloc 的含义。）

8. 改进我们的 Block 类，以使得它可以和有着构造函数但没有缺省构造函数的对象一同工作。

9. 你希望往 List 模板中增加什么函数以使得它可以成为在你的项目中所使用的工作代码？请对你的选择进行解释。你所新增的函数会对性能带来什么样的影响？对程序的可理解性又带来什么样的影响？

10. 在我们的 List 类中，当 List 被摧毁时我们并没有去更新所有的迭代器。对于每个 Listiter 中的成员函数，请为它们选择当 List 被摧毁后函数被调用时的语义，并实现你的选择。

11. 除了导入 Listiter::at_end 函数（见 8.5.2）之外，我们还可以试着通过一个引用参数返回一个整数，当我们到达 List 的末端时，它会被置为 0：

```
template <class T>
class Listiter {
public:
// ...
    int next(T&);    // 常规形式
    T   next(int&)   // 没有 T 的缺省构造函数时的形式
};
```

上面的设计有什么问题吗？

12. 在 USL 标准组件库中还包含了一个 Set_of_p 模板，Set_of_p<T>在本质上来说就是一个 Set<T*>。请解释为什么这样做会比较好。

13. 我们对于 List<char>的特化版本有着如下的特性：对于链表的插入和删除都将导致链表的重新排列。和通用的 List 模板类相比，这样做的潜在不兼容性在什么地方？

14. 设计和实现一个 Listiter<char>的特化版本，以使得它可以和我们的 List<char>特化版本一同工作。

第9章

重　用

重用是一种用来提高编程效率的最佳方案：不用去编写新的代码总是要比编写新代码来得快，不是吗？

然而，要做到那样并不容易：重用不只是会给我们带来好处，也会产生更多的开销。要想对代码获得成功的重用（如：我们从库中调用一个函数或者使用库中提供的类），那么我们从中所获得的好处必须要大于它产生的开销。那些开销包括：

- 寻找——用户必须从一堆可供使用的库中选出合适的库来；
- 获取——用户必须在他/她的系统上拥有一个该库的版本；
- 学习——用户必须理解该库所带的抽象模型，并熟悉其详细接口；
- 控制权的丢失——用户不再具有对代码进行修改以使得它“正好”可以适合我们的应用这个权利[1]。如果有两个彼此交互很紧密的库，这个问题就将变得愈发难以解决；
- 出错的效率变差——如果在项目中找到了 bug，用户将不得不和代码的提供者一起合作来进行除错活动。如果我们没有其他的替代方案，这个问题将会变得十分严重；
- 风险——如果我们使用的库被证明是有问题的，或者速度很慢，或者没有应有的技术支持，那么我们就可能不得不将它从我们的项目中移除并重新亲自实现它。当开发进行到后期时，这就可能会导致项目的失败（而不仅仅是给项目组成员的挫败感）。

更糟糕的是，开发人员通常都会低估构建自己所需使用的库所需的开销。我们必须保证，在准备重用时，可预期的好处必须要大于可预期的从头构建库所需的开销。

然而，重用在可行时仍然是一件让人兴奋的事情。本章将就“该如何在最小化开销时也取得收益”这个问题进行讨论，并将着重讨论下面这些主题：

- 发现 - - 如何让潜在的用户轻易地发现我们的库，并决定是否采用它；
- 健壮 —— 如何使得我们的代码通过严格的考验，使得它能够以我们甚至不能预期的方式工作；
- 名字冲突—— 如何来减少发生在我们库所提供的外部名字（external name）和程序中其他部分所定义的外部名字之间的名字冲突；

1 对于这个问题一直存在着争议，有人认为这样做会给项目带来好处；然而做出重用决策的人在开始是不能预见到这一点的。

● 性能 —— 如何来分析和调整我们库运行时所需的空间和时间。

存储管理是导致程序 bug 和性能问题的一个常见来源，我们将会在讨论有关程序健壮性和性能的小节中对此进行讨论。

9.1 发现和获得

程序员应该能够尽快地判断一个特定的库是否是有用的。如果是的话，那么它应该能够很容易地被设置并运行。

9.1.1 简单的抽象

使用我们库的潜在用户可能只会通过翻阅我们提供的手册中的第一行就做出是否使用它的决策。所以我们应该尽可能地用一行简短的总结性的文字来概括我们的库。如果我们做不到这点，那么原因可能就是：库太特殊化了；或者库中包含有太多的抽象。不管是哪种情况，它都会导致库不适合被挑选到我们的应用程序中使用。

9.1.2 文档

当然，我们还必须提供一份简洁的手册，它里面必须包含有所有有助于让人决定库是否可以被使用的信息。决定是否使用库的人不可能会花时间去在寻找库的过程中阅读完整的教程，因此我们的手册应该是用来介绍那些小规模的库（如 String 类）。对于一个实际中使用的库（如完整的输入/输出包）来说，我们最好是提供一份详细的教程。

9.1.3 精心挑选的类名

为我们的类挑选一个合适的名字是设计库的过程中最困难的一部分。我们设计的类几乎总是要被其他人使用的。两个开发人员经常会因为一个定义模糊、不够精确的类名而导致对同一个抽象具有不同的理解。

在我们的文档中，类的名字是其中最重要的部分。我们应该在设计过程中不断对其进行完善。CRC 卡片（参见 1.4 节）所带来的一个好处就是：我们可以尽可能早地告诉其他人我们设计的类的名字；这样，如果有人对我们给出的类名有着不同的见解，我们也可以在设计的前期过程中即时地发现它。

9.1.4 获取

小规模的类必须很容易被获取——不能即时被获取的库也就不可能被用户所使用。这就意味这我们应该将那些小规模的库放到一起地发售，使得购买者能够决定是否预先购买整个集合，以便于今后的使用。

9.2 健壮性

显然，能够被重用的库必须尽可能少地出现bug；因为处理其他人编写的代码中的bug的成本会比较高昂。但是，健壮性并不仅仅意味着很少的bug。一个能够被重用的库还应该可以处理作者未能预见的使用情况——成功的库总是经常会被用户以作者从未想过的方式来使用的。

我们先来考虑如下的String类的实现：

```
class String {
private:
    char*   data;
    unsigned char length;
    // ...
```

我们用数据成员length来保存字符串的长度。在大多数程序中，上面的那个实现都可以很好地工作，但当我们用一个长度超过unsigned char所能容纳的范围的字符串来创建String对象时，上面的类就将会有问题产生。向外声称某些事情“永远不会发生”是一种天真的做法；有的用户可能会使用一个String对象来存储一整个文件的内容。这样做并没有什么问题，但我们上面给出的实现却不能满足这个要求。

避免对对象的大小进行假设是用于提高我们代码健壮性的一个重要途径。下面我们还将给出其他的几种方法。

9.2.1 断言

在C++中，assert宏甚至比它在C中的重要程度还高。缺省情况下，assert会带有一个参数，当该参数等于0时，它会终止程序并向外给出一条诊断信息：

```
Node *p = something();
assert(p);
Node* kid = p->child(0);
if (kid)
    assert(kid->parent() == p);
```

如果我们定义了NDEBUG宏的话，我们就可以把编译单元中的所有assert都给关闭掉：

```
CC -c -DNDEBUG file.c
```

当我们按照上面的方式做了之后，所有的assert都将变成空操作。这意味着我们可以“屏蔽”掉断言，使得当定义NDEBUG后，它们不会影响我们程序最终版本的性能。

面向对象的程序和基于过程的程序相比，它们的结构并不相似，这也导致断言的大量使用变得愈发重要起来。当基于过程的程序崩溃时，我们可以很轻易地通过对当前函数的回溯找到发生错误的原因。而对于面向对象的程序来说，这样做就要困难多了，因为相对基于过程的程

序来说，它里面的函数通常要小得多，并且程序的控制流程也要复杂得多（尤其是当我们使用了虚函数）。这就使得我们不得不尽可能快地在程序出错的时候将程序终止；如果在程序结束前它还继续运行了一段时间，那么我们就很难找出到底是程序的哪部分导致程序出错。

我们还可以使用断言来作为库中的前置条件校验。如果我们在文档中指出一个函数的参数不能为 0，那么我们在该函数中所做的第一件事情就是使用 assert 来判定该参数是否为 0。如果用户向函数中传递了一个 0 作为参数，那么我们就可以即时地在函数调用处获得一个断言错误，而不是在我们库中的某个地方得到一个程序崩溃的结果。

断言和异常

我们应该在 assert 中向外抛出异常，而不是打印一条出错信息，然后程序就转储内核。ISO/ANSI C++标准委员会现在也正在研究这个话题，如果断言中抛出异常的方法被大量支持的话，我们最好是使用它们，因为这样做可以让用户有机会来捕获（并处理）发生在库内部的错误。

assert语句中的副作用

在 assert 中，对于可以正确运行的操作来说，其操作数不能有任何的副作用：

```
// 当操作失败时，引发一个断言错误
assert(release_lock() == SUCCEED);
```

当我们定义了 NDEBUG 时，上面的测试将会被忽略，这也将导致我们不会去调用 release_lock()！

一些不能使用assert的错误检测

通过使用 NDEBUG，我们可以在编译时去除断言；但程序中还是有着一些错误检测是我们不希望去除的——不管我们有没有定义 NDEBUG。例如：

```
char* space = new char[BUFFSIZE];
assert(space);
```

即使彻底的测试过程可以给以我们足够的信心来去除掉用于检测逻辑错误的断言，我们还是无法保证程序是否会碰到内存不足的情况。除非我们的存储分配器被设计成当碰到内存不足的情况时向外抛出异常（或者进行某些动作来获取一些内存），否则我们总是要进行上面那样的检测。当需要进行无条件的检测时，我们很容易想到使用一个函数（而不是断言）来处理它：

```
inline void check(int expr, const String& diag) {
    if (expr == 0) {
        cerr << "Internal error: " << diag << endl;
        abort();    // 或者抛出异常
    }
}
```

assert 和 check 的实现方式并不相同。在 assert 宏中，我们使用了_FILE_和_LINE_这两个预处理器中的宏，并用它们来构造包含有文件名和行数信息的诊断信息。对于 check 来说，这

种做法并不可行：因为它是一个函数，而不是宏，我们能够得到的文件和行数的信息总是会等同于 check 函数体的定义处的信息，而不是我们期望的 check 被调用处的信息。另外，我们在 check 中还使用了一个 String 参数来作为检测失败时的诊断信息。

当然，在需要的前提下，我们还是可以使用_FILE_和_LINE_这两个宏来构建我们的诊断信息的。

9.2.2 避免使用固定大小的数组

在很多的 C 程序中都使用了固定大小的数组，这使得它们在输入过大的情况下很容易崩溃。C 程序员使用固定大小的数组的原因是：使用 malloc 和 free 很不方便，并且还容易出错；对于 C++程序员来说则没有这种借口。除非知道数组在多大时就已经足够大了，否则我们就应该使用容器来代替数组。

例如，下面对数组的使用就显得极为合理：

```
const char* days_of_week[7] = {
    "Sunday",
    "Monday",
    "Tuesday",  // ...
```

这是因为我们都知道，一周内的天数是一个固定的常量，并且也不太可能发生改变。但是，让我们来考虑下面的这个程序：它通过标准输入读入职员的名字，并把它们放到一个有着 MAX_EMPLOYEE 个 Employee 对象的数组中去：

```
const int MAX_EMPLOYEE = 1000;
Employee* employee[MAX_EMPLOYEE];

void read_employees(istream& in) {
    Employee** next = employees;
    String name;
    while (in >> name) {
        if (next >= &employee[MAX_EMPLYEE-1])
            internal_error("Too much input");
        *next++ = new Employee(name);
    }
    *next = 0;
}
```

上面代码中的 MAX_EMPLOYEE 的值可能是我们从某处取来的，它是我们在对“是否让数组大到可以容纳所有可能的输入”以及“是否让数组尽可能小到不要使用过多的内存空间”之间得到的一个中值。如果这个库被广泛使用的话，那么就有可能会出现下面这种情况：用户将它用于整个联邦政府的薪水统计中；这也就将导致程序崩溃。

上面程序中的另外一个不足之处就在于：不管实际读入的职员个数有多少，它都使用了

MAX_EMPLOYEE * sizeof(Employee*)个字节的内存来容纳所有指向职员的指针。我们可能会期望在程序中我们只需为实际读入的职员分配内存，而不是为可能出现的最大输入分配内存。

下面就是一种可行的做法。我们在此使用了第 8 章所定义的 Block 类，但是我们还可以用其他等同于数组的容器来代替我们使用的 Block：

```
Block<Employee*> employees;
int top_employee = 0;

void read_employees(istream& in) {
    String name;
    while (in >> name) {
        employees.reserve(top_employee);
        employees[top_employee++] = new Employee(name);
    }
}
```

这个新版的 read_employees 相对前面的那个版本来说要更简单，同时其健壮性也更好，只有当程序消耗掉所有可用内存时，它才可能出错。虽说使用 Block 会导致我们读入数据的时间变长，但一旦我们把所有的数据都读入到 Block 中后，对它的使用就将变得和使用数组一样快了。事实证明，在我测试所用的计算机上，主要的运行时间都被消耗在 new 操作上，所以上面两个版本的 read_employees 之间的性能区别几乎可以忽略不计。

9.2.3 为程序提供调试版本

对于某些重要的函数和类来说，为它们提供调试版本是另一种提高代码健壮性的方法。通过使用调试版本中大量且广泛的检测（虽然这样会对性能产生影响），我们就可以尽可能早地发现程序中的错误。在很多的情况下，我们通常都是对程序中的主要数据结构进行完整地检测。例如，下面的函数就是用来对一个双向链表进行遍历，以确保其间的指针都可以指向正确的元素：

```
template <class t>
void Dllist<T>::audit() {
#ifndef NDEBUG
    Dllist_link_base* ptr = head;
    while (ptr && ptr->next)
        assert(ptr->next->prev == ptr);
#endif
}
```

我们在任何会改变链表的操作前和操作结束后都会去调用这个 audit 函数：

```
template <class t>
void Dllist<T>::insert_after(const T& t) {
    the_list->audit();
```

```
    // 插入
    the_list->audit();
}
// ...
```

上面的代码就可以确保，任何会对 next 和 prev 指针产生不适当赋值的操作都会马上触发一个断言错误。在 audit 函数中，它的运行时间和链表的长度成正比，但这并不会给我们带来什么影响：因为我们只是在开发过程中才使用它。

我们必须确保 audit 函数在文档中被很好地定义，以使得不会有某些天真的用户在发售他们的程序时，程序中仍然存在着 audit 的代码。另外，我们还必须意识到，使用了 audit 会改变我们程序的复杂度。如果在遍历链表的过程中，我们在每次步进时都调用了 audit，那么这个遍历的时间复杂度就将变为 $O(n^2)$。如果使用 audit 导致的开销变得过大的话，我们就应该考虑是否禁止使用 audit——即便是在开发过程中也是如此。

9.2.4 Linton 法则

带有一个引用参数的函数可能会在函数中使用到该参数的地址，并把它存储在某个即使在函数返回后仍然都存在的地方。对于调用者来说，知道哪个函数会这么做是很重要的一件事——因为调用者必须关心该操作是否会导致被存储的指针变为空悬。例如：

```
const Thing* last_accessed_thing;
void access(const Thing& t) {
    // ...
    last_accessed_thing = &t;
}

extern Thing get_a_thing();
void f() {
    access(get_a_thing());
        // last_accessed_thing 现在成为了空悬指针
}
```

在上面的代码中，由 get_a_thing 返回的临时对象在 access 返回时，会被编译器自动摧毁，这就将导致 last_accessed_thing 指针变为空悬指针。

为了避免出现这种情况的 bug，Mark Linton 提出了如下的编程法则：

函数中不应该将指向其引用参数的指针保存在函数返回后还可以访问到的变量中。如果需要这么做的话，我们就应该使用指针参数。

上面的法则很有意义。在函数调用中传递引用参数的写法和传递值参数的写法是一样的；对于程序员来说，这会使得他们很容易忽略掉这两者间的不同。如同我们在 2.7.1 节中所见的那样，类的设计者可以通过使用常量引用来代替应该传递值的参数，以此来得到性能上的提升。这种做法也会使得程序员可能会忘了某些情况——如在被调用的函数使用了某种方式来“记

录”它的常量引用参数。与此类似的是，如果碰到的是引用参数，程序员就会认为它的值会在函数调用的过程中被修改。如果我们在调用函数时使用了引用参数，结果它的值的改变发生在函数返回之后，这就很容易引起别人的惊讶。

如果在 access 中使用了 const Thing*，那么我们的调用就将变为：

```
extern void access(const Thing*);  // 存储指针
extern Thing get_a_thing();

void f() {
    Thing t = get_a_thing();
    access(&t);
}
```

上面的代码仍会导致 last_accessed_thing 变为空悬，但一旦程序由于空悬指针而崩溃时，程序员就很可能会去怀疑这段代码。

当我们很难从代码中发现函数中存储的指针时，Linton 法则就显得更为重要了。如果我们能够很轻易地发现函数中存储了引用参数的指针，那么我们就可能会因为美化代码而去破坏 Linton 法则。例如，我们在第 8 章中定义的 Listiter 模板的构造函数就使用了一个 List&参数，并且在被构建的 Listiter 中存储了一个指向该 List 的指针。由于创建迭代器的行为十分常见，所以我们应该将该创建操作尽可能地变得简洁。通过使用引用参数，我们就可以这样编写：

```
List<int> iter(list);
```

而不是这样

```
List<int> iter(&list);
```

由于迭代器是对于指向容器内元素指针的抽象，所以“迭代器会‘记忆’它所指向的容器”这件事情并不会让阅读代码的人觉得惊讶。

9.3 内存管理

语言的设计者对于内存管理的看法可以分为两类。有的认为内存管理十分重要，它应该被交与程序员来处理；另外一些人也认为它很重要，但是正是由于它很重要，它不应该被交与程序员来处理。C++（以及 C）采用的是第一种做法。这使得程序员有着最大的弹性来处理该问题；但一旦我们搞乱了一些简单的顺序，原本安全（并且通常不会引人注意）的存储操作也将会导致程序崩溃，或者导致程序执行一些错误的操作。

C++中还包含了一些用于帮助类设计者和用户避开存储问题的机制。然而这些帮助机制并不会主动出现在我们面前；我们必须明确地去使用它们，以此在我们设计的类中减少有关存储的问题。

在本节中，我们会展示一些用来保证正确删除的方案。这些方案中的核心思想就是让计算

机以某种方式来记录被分配的内存，从而程序员就不需要时刻记着去删除它们。

9.3.1 使用构造函数和析构函数来管理内存

最安全的方法就是在构造函数中使用 new，在析构函数中使用 delete。C++中的语言规则确保[2]了所有的自动对象和全局对象都会被正确地摧毁。C 风格的做法（让程序员去处理何时释放内存）则很容易导致 bug 的产生。考虑如下的函数，它会将职员姓名中的多个连续的空格替换成单个的空格：

```
void enter_employee(const char* name) {
    char* canon = new char[strlen(name)+1];
    char* cp = canon;
    char last = 'X';
    while (*name) {
        if (!(*name == ' ' && last == ' '))
            *cp++ = *name;
        last = *name++;
    }
    *cp = '\0';
    employee_database.enter(canon);
    delete[] canon;
}
```

上面的代码可以正常的工作，但我们很容易就将 bug 导入到它里面去。例如，假设我们现在在姓名中使用了‘@’字符来代表“不在名册中的”职员，并且将代码修改为如下形式：

```
void enter_employee(const char* name) {
    char* canon = new char[strlen(name)+1];
    char* cp = canon;
    char last = 'X';    // 也可以是任意的非空格字符

    while (*name) {
        if (*name == '@')   // 是否在名册中？  *新增代码*
            return; //                          *新增代码*
        if (!(*name == ' ' && last == ' '))
            *cp++ = *name;
        last = *name++;
    }
    *cp = '\0';
    employee_data.enter(canon);
    delete[] canon;
}
```

上面对于‘@’的检测给代码中导入了一个 bug：如果函数在那返回的话，函数中已分配

[2]事实上是“部分确保”。在 9.6 节中我们会给出一些反例。

的内存就将丢失。

如果我们使用了对象来管理内存的话，这样的 bug 产生的可能性就将大为减低。在下面的示例代码中，我们使用了 String 类来代替 char*：

```
void enter_employee(const String& name) {
    String canon;
    char last = 'X';    //也可以是任意的非空格字符
    int len = name.length();

    for (int i = 0; i < len; ++i) {
        char ch = name[i];
        if (ch == '@') // 是否在名册中?
            return;
        if (!(ch == ' ' && last == ' '))
            canon.append(ch);
        last = ch;
    }
    employee_data.enter(canon);
}
```

在此，内存是由 String 的实现来维护的。无论何时程序的控制流程转到该函数之外（可能是函数返回，也可能是函数向外抛出异常），所有已有的自动对象（在本例中是 canon）都会被正确的摧毁，这样也就避免了内存泄漏。

9.4 可选的内存分配方案

使用自动对象来处理内存的释放只适合于同一个函数中的内存分配和释放。然而我们不可能总是碰到这种情况下的内存分配和释放；在本节中，我们将会来考虑其他的一些常见的内存管理方案，它们甚至可以工作在“分配和释放不在同一个函数中”这种情况下。

9.4.1 使用计数器的智能指针

在第 7 章我们就已经详细地讨论了这种方案：我们用智能指针对象来管理一个计数器，当计数器变为 0 时，我们就会删除该对象。

9.4.2 通过记录所有的指针和对象来实现的垃圾收集机制

在另外一种备选方案中，我们关注的计数对象包括对所有的（智能）指针以及被它们所指向的对象。每隔一段时间，我们就会使用一个“垃圾收集器”去检测所有已有的指针，并对它们所指向的对象进行标注。如果存在未被标注的对象，它就会将该对象删除。

这种做法既有好处，也有坏处。虽说对一个使用了计数器的指针进行的修改涉及到多个计数器的改动（参见 7.4.2 的代码），对一个被收集的指针进行修改速度还是和对一个一般指针进

行修改的速度是一样的。在天平的另一端，对指针以及被指向对象的创建和删除则要费时得多，因为这些操作必须要更新用来记录已有指针和对象的数据结构。应用程序的设计者必须决定在什么时候来运行垃圾收集子程序。对于实时系统来说，在一个请求被响应前，我们可能得不到足够的时间来完成我们的垃圾收集操作，这也会使得我们的垃圾收集器的逻辑变得复杂起来。

然而，如果程序中的指针的值经常发生改变，而指针本身和对象并不会经常被创建和摧毁，那么使用这种方案就可以给我们带来最佳的性能。

对于那些外界不能获得的循环指向对象来说，我们还是无法收集它们。不过我们可以通过对垃圾收集器进行改进，增加一个根指针来应付这种情况。如果所有的“活着的”对象都是要么被根指针所指向，要么被另外一个“活着的”对象中的指针所指向，那么垃圾收集器就可以从根指针开始对所有后续的指针进行遍历，为每一个被指向的对象进行标注。由于垃圾收集器必须能够找到对象中的所有指针，我们很难构建一个通用的库来支持这种垃圾收集方式。在实际应用中，我们通常的做法是：为某个类手工创建一个单独的函数来处理它。

9.4.3 Arena

arena 是用来管理内存的最简单方案之一，但是它们只适用于某些特定的应用程序。在这些程序中，每个对象都是在 arena 中创建的。在后期，当 arena 被清空时，所有在 arena 中的对象都会被删除。

例如，我们假设有一个 C++的解析器，它会去解析一系列的文件，在某个文件中都会包含一系列的外部声明。在解析过程中，解析器可能会创建一系列的符号表用来表示解析中碰到的名字。解析器的作者可能会认为，函数中声明的符号在离开函数后就将变得没有价值，可以被丢弃了；但对于有着文件范围属性的符号来说，在文件被完整解析之前，它都不能被丢弃。我们可以用两个不同的 arena 来分配这两种不同的符号：函数级的 arena 在离开函数后就会被清空；文件级 arena 的清空则发生在文件被处理后。

由于 arena 中的逻辑和它里面存储的对象无关，这样我们就可以将它实现为模板。在 arena 中，我们维护了一个集合用来容纳指向它所存储的对象的指针：

```
template <class T>
class Arena {
private:
    Set<T*> contained;
public:
    Arena() {}
    ~Arena() { empty(); }

    void enter(T* p) { contained.insert(p); }
    void empty();
};
```

如果对象希望被放入 Arena，它就会去调用 Arena 的成员函数 enter，并将自身作为参数传

递给该函数。我们可以在任意时刻来显式地对 Arena 调用 empty，以此来摧毁 arena“中”所有的对象：

```
template <class T>
void Arena::empty() {
    Setiter<T*> iter(contained);
    T* tp;
    while(iter.next(tp)) {
        contained.remove(tp);
        delete tp;
    }
}
```

为了安全起见，在 Arena 的析构函数中我们也调用了 empty。

显然，由于对象是被 arena 给删除的，所有使用 arena 的对象的创建都必须使用 new 来进行，并且我们不能将一个对象放入到多个 arena 中去，也不能在其他地方对它进行删除（参见问题 6）。

delete的操作数

在 Arena<T>::empty中还存在着一个难以发现的编程陷阱：我们在删除指针之前必须先将它从 contained 中移除。根据 C++中的规则来说，如果 delete 的操作数是一个可修改的左值，那么在被删除后，它将获得 一个未定义的值。该规则使得编译器实现可以在删除 tp 后对它的值进行修改（通常都是将它设置为 0），以捕获如下这种经常会出现的 bug：

```
// ...
delete ptr;
return ptr->foo;
```

但是这也意味着我们不能指望 tp 在被删除后仍然有着有意义的值。下面的做法就将导致一个 bug 的产生：

```
template <class T>
void Arena::empty() {
    Setiter<T*> iter(contained);
    T* tp;
    while(iter.next(tp)) {
        delete tp;
        contained.remove(tp);   // Bug!
    }
}
```

在某些编译器实现中，delete 并不会去改变 tp 的值，因此上面的代码可以幸运地在这些编译器实现中运行；但一旦我们将它移植到那些会在 delete 后将 tp 置为 0 的 C++实现上时，上面的程序就将崩溃。

类Symbol

现在我们就可以使用我们的 Arena 模板来存储我们编译器的符号表了。Symbol 的构造函数中的一个参数就是用来指示 Symbol 所在的 arena：Arena<Symbol>。（注意：所谓的“在” arena 中只是一个抽象中的观念；实际上我们还是在堆中创建 Symbol。）当我们构造 Symbol 时，我们都会通知它所在的 arena：

```
class Symbol {
// 此处忽略细节
public:
    Symbol(Arena<Symbol>&, const char*);
    ~Symbol();
};

Symbol::Symbol(Arena<Symbol>& arena, const char* d) {
// 先构造 Symbol 对象，然后...
    arena.enter(this);
}
```

我们先前的假设是在 Symbol 的构造函数中用参数的方式来传递它所处的 arena；实际上我们还可以使用其他方法（如使用一个全局变量或者根据 Symbol 的内容来选择合适的 arena）。

9.5 传递参数给 operator new

用来容纳 Symbol 的 arena 并没有告诉我们 Symbol 设计存储的地方，而是告诉了我们一些有关对象抽象行为的事情：它可以用来决定对象何时被摧毁。与大多数用来指定行为的信息一样，我们将 Arena 作为参数传递给了构造函数。

如果我们希望控制对象到底是存储在什么地方，C++也可以允许我们传递参数给 new 操作符（参见后面的“复习”）。由于 operator new 只是简单地获取内存（它不会去初始化对象）我们就可以使用它的参数来设置对象被存储的地方；否则的话，我们就应该在构造函数中使用这些参数，而不是在 operator new 中使用。

当应用程序运行在某个有着有限快速内存的平台上时，我们就可以考虑使用 placement 语法。当对象被创建时，我们可以向 operator new 中传递一个额外的参数，这可以使得我们把对象创建在快速内存中：

```
extern void* malloc_fast_memory(size_t);   // 获取快速内存
enum Memory_kind { fast_mem, slow_mem };

void* operator new(size_t bytes, Memory_kind kind) {
    if (kind == fast_mem)
        return malloc_fast_memory(bytes);
    else
```

```
        return malloc(bytes);
}
```

下面的调用就是用来获取快速内存的：

```
Thing* important = new (fast_mem) Thing;
```

复习：Placement 语法

通过在紧接着 new 关键字后使用括号列表，用户可以向 operator new 传递参数：

```
Memory_info info;
Thing* tp = new (info) Thing;
```

这些参数被作为额外的参数传递给 operator new，它们被放置在隐藏的第一个（类型为 size_t，用来指示要分配的字节数）参数后。我们必须提供一个 operator new（无论是和类相关的还是全局的）来使用这些参数。在我们前面给出的例子里面，必须有这么一个 operator new，它的第一个参数的类型为 size_t，第二个参数的类型为 Memory_info：

```
void* operator new(size_t bytes, Memory_info info) {
//此处忽略细节
}
```

某些 C++编译器会包含（在 new.h 中）一个特殊的 operator new，它被用来指定被创建对象的存储位置：

```
void* operator new(size_t, void* where) {
    return where;
}
```

上面这种形式的 operator new 就可以强制将我们新创建的对象放置到某个特定的地方：

```
#include <new.h>
main() {
/* 在地址 0x1234 处构建一个 Thing 对象 */
    new ((void*)0x1234) Thing;
```

上面的代码会在地址 0x1234 处创建一个 Thing 对象。使用该函数的用户最好得清楚它的含义。

9.5.1 Placement 语法与 operator delete

虽然我们可以传递参数给 operator new，但我们还是无法传递参数给 operator delete。如果 delete 操作符需要了解内存是怎么被分配的，它只能从它那惟一的参数（类型为 void*）中去

获取。通常我们都是通过维护一个单独的数据结构来做到这一点的：

```
static Set<void*> allocated_fast_memory;

void* operator new(size_t bytes, Memory_kind kind) {
    void* vp;
    if (kind == fast_mem) {
        vp = malloc_fast_memory(bytes);
        allocated fast memory.insert(vp);
    }
    else {
        assert(kind == slow_mem);
        vp = malloc(bytes);
    }
    return vp;
}

void operator delete(void* mem) {
    if (allocated_fast_memory.contains(mem)) {
        allocated_fast_memory.remove(mem);
        free_fast_memory(mem);
    }
    else
        free(mem);
}
```

请注意：如果使用的是缺省的 operator new，上面的代码就可以正确工作：

```
Thing* tp = new Thing; // 没有使用 placement 语法
delete tp;
```

我们应该对使用这种内存分配方案的程序进行分析，以确保从快速内存中得到的收获不会因为额外的维护那个 Set 的开销所抵消。

当然，如果通过检测指针的值我们就可以判断它是不是指向快速内存（有可能所有的快速内存都存在于某个已知的地址区间中），那么那个额外的数据结构就将变得不再是必要的了；通过检测那个 void*参数，operator delete 就知道该怎么做了。

9.6 管理外部资源

我们可能会试图使用对象来管理其他资源，甚至是那些位于运行着的进程之外的资源。例如，我们可能会用一个对象来表示数据库中针对某个部分的锁；当对象被摧毁时，我们同时也就完成了解锁操作。如果真的这么做的话，我们必须时刻记住，世界上还有很多种方法来结束一个进程并且还不会去调用那些已有自动对象的析构函数。

在当前的语言规则中，exit()会导致静态对象被摧毁，但它不会去摧毁那些自动对象！如果我们依赖于自动对象的析构来释放进程中获取的外界资源（如数据库锁）我们就有可能陷入麻烦之中。其他的一些系统调用也会导致该问题的产生。在 UNIX System V 以及类似的操作系统中，对于 exec()和 abort()函数的调用也不会导致自动对象被摧毁；而对于一个没有跟在 exec()后的 fork()调用则会导致自动对象被按位复制，这样就导致外界资源被释放两次！

9.7 寻找有关内存的 bug

即便我们再努力，我们早晚还是要去调试有着内存问题的代码：问题有可能是内存泄漏，多次删除同一块内存，或者是使用以及被删除的内存。在本节中，我们将讨论一些常用的用来检测和修正这些 bug 的技巧。

9.7.1 在析构函数中进行无意义的破坏

对于那些涉及到使用已经被删除的内存的 bug，它们的处理尤为麻烦，因为这样的程序只要运气好，经常（而不是一直）都能够工作。只有在内存分配器重新分配那些仍被使用的内存时，它们才会出现。如果在删除和（无效的）后续使用之间的间隔很小，那么内存分配器对该块内存的重新分配的几率也会很小（但不会是 0!），这也使得我们很难重现该 bug。

一种用来捕获这种 bug 的做法就是在每个析构函数中毫无理由地对该对象所使用的内存进行随意填充。这样即使内存分配器并没有将这块内存重新分配，我们在后续对于该对象的使用时也可以即时地捕获这些 bug。例如：

```
class Employee {
private:
    String name;
    int salary;
public:
    //此处忽略细节
    ~Employee();
};

Employee:~Employee() {
    salary = -1; // 或者其他的明显的无效值
}
```

如果原来的程序中有一个 bug 是使用了一个已经被摧毁的 Employee，那么现在它就可以作为一个明显且可以重现（例如我们对 salary 进行检测，看其是否会等于 $-1）的问题被立即找出来，而不再是原来的偶尔出现、难以重新的程序崩溃现象。

在我们的示例中，当 Employee 被摧毁时，它里面的成员 name 也会被 String 的析构函数所摧毁。如果在那个析构函数中也会对 name 进行随意填充，那么 Employee 就不需要去担心

它会有什么问题，Employee 所需的只是在自己的析构函数中对那些非对象的成员进行随意填充。

如果对成员的填充所需的运行时间过长的话，我们可以选择条件编译的方式来去除它：

```
Employee:~Employee() {
#ifndef NDEBUG
    salary = -1;
#endif
}
```

9.7.2 提供自定义的 new 和 delete

另一种技术包括使用一个特殊版本的::operator new（它可以记录它所返回的每个指针）以及一个对应版本的::operator delete——如果它的操作数不是经由前面的::operator new 返回（或者操作数已经被删除了多次）的话，它就会向外提交一个错误信息。

听上去好像很简单，不过我们还将碰到一个问题：我们的::operator new 实现不能用于任何的会去调用::operator new 的自定义类型。由于大部分的容器类使用的都是 new 和 delete，我们就可以减少编写 C 风格的、使用了 malloc 和 free 的代码。我们的::operator new 会去分配内存，并在返回前将结果存储到一个指针数组中去。在这个子程序中，大部分的逻辑都是用来处理指针数组在必要时的增长：

```
#include <stdlib.h>
#include <assert.h>

/* 为简单起见（我们只是将它用于除错过程中！），忽略掉有关 new_handler 的问题 */

static void** active_ptrs = 0; // "block" of outstanding ptrs

static int active_ptrs_l = 0;  // active_ptvs 的大小
static int used_ptrs = 0;      // 非零条目的数量
const int chunk_size = 100;// 增长速度有多快

extern void* operator new(size_t size) {
    // 获取所请求的内存
    void* mem = malloc(size);
    assert(mem);

    // 记录指针。不能使用那些在实现中会使用 operator new 的对象

    if (used_ptrs == active_ptrs_l) {
        // 数组需要增长

        int new_size = active_ptrs_l + chunk_size;
```

```
            if (active_ptrs)
                active_ptrs            =            (void**)realloc(active_ptrs,
new_size*sizeof(void*));
            else
                active_ptrs = (void**)malloc(new_size*sizeof(void*));
            assert(active_ptrs);

            // 将那些新的指针入口置0

            for(int i = active_ptrs_l; i < new_size; ++i)
                active_ptrs[i] = 0;
            active_ptrs_l = new_size;
        }

        // 寻找用来存储指针的地方

        for(int i = 0; i < active_ptrs_l; ++i) {
            if (active_ptrs[i] == 0) {
                active_ptrs[i] = mem;
                ++used_ptrs;
                return mem;
            }
        }

        abort();    // 说实话，这种现象不可能发生
        return 0;
    }
```

请注意在函数末端所使用的 abort()。将 abort 放置在我们认为函数中不可能到达的地方是一个好的习惯。如果我们错了，或者对代码的某些后期改动使得程序可能运行到那个地方，那么我们所设置的 abort 就会被触发。

::operator delete 中会检测其参数以确保它是从上面的::operator new 中所返回的：

```
    extern void operator delete(void* v) {
        // 删除"0"指针是可行的
        if (!v)
            return;

        for(int i = 0; i < active_ptrs_l; ++i) {
            if (active_ptrs[i] == v) {
                active_ptrs[i] = 0;
                --used_ptrs;
                free(v);
                return;
            }
```

```
        fprintf(stderr, "Delete of object that was not newed\n");
        abort();
    }
```

在C++语言规则中，对0指针进行删除是合法（且无害）的操作，所以我们的代码中也应该允许出现这种情况。

代码中的线程搜索意味着我们的内存分配器不够高效，但由于我们只是把它用于调试过程中，所以这样做没有什么问题。

我们同样还可以使用下面这个子程序来打印当前内存使用情况的信息：

```
extern "C" void print_allocated() {
    if (used_ptrs) {
        fprintf(stderr, "Allocated memory:");
        for(int i = 0; i < active_ptrs_; ++i)
            if (active_ptrs[i])
                fprintf(stderr, "%#x\n", active_ptrs[i]);
    }
}
```

上面的函数会打印程序中当前每块正被使用的内存的首地址。如果我们能够在程序退出前、所有的静态对象被摧毁后调用print_allocated函数，我们就可以用它来帮助查找内存泄漏。至于如何做到这一点，这依赖于我们的编译器和操作系统。注意，我们将该函数声明为extern "C"：这使得它可以被用C或汇编编写的系统例程所调用。

我们还可以使用自定义的new和delete来捕获另外一种常见的bug。有着这种bug的程序通常都可以很运行地工作，但当用到用户提供的内存分配器或移植到其他编译器时，它就会崩溃。例如：

```
class String {
    char* rep;
public:
    String(const char* = "");
    ~String();
//此处忽略细节
};

String::String(const char* cp)
: rep(new char[strlen(cp)+1]) {
    strcpy(rep, cp);
}

String::~String() {
    free(rep);  // bug!
}
```

每个 new 都必须有一个相应的 delete。而在上面的代码中我们使用了 free；如果在::operator new 中调用了 malloc，那么它就可以（幸运地）工作。对于大多数的 C++实现来说，缺省情况下::operator new 都会去调用 malloc；当如果我们把::operator new 替换成另外一个不调用 malloc 的版本时，上面的代码就会崩溃。（记住：任意的用户，而不仅仅是 String 的提供者，都可以在链接期中为它提供一个::operator new！）对于 print_allocated 的调用可以帮助我们找到这个 bug——因为所有没有用 delete 释放的内存都将显示为一个内存泄漏的 bug。

如果分配时使用了 malloc 而释放时使用了 delete，我们也能够得到一个类似的 bug；在本例中，在 delete 被执行完后，我们提供的函数就可以捕获住这个错误。

9.7.3 数组的删除

根据 C++的语言规则，当我们删除数组时，必须使用 delete[]这种语法形式。其中的[]就会通知编译器产生用来计算数组大小、并逐一调用数组中每个元素的析构函数的代码。如果我们没有给出[]，那么只有数组中的第一个元素会被摧毁。

如果数组元素是内建类型，由于它们不存在析构函数，在大多数（目前我还没有碰到过有问题的）编译器上面，缺少[]的代码也可以幸运地正常工作：

```
void main() {
    char* cp = new char[2];
    delete cp;
}
```

即便上面的代码可以幸运地正常工作，在语言规则中它的行为还是未定义的！如果缺少[]的话，不管其元素类型如何，对数组的删除都将会得到未知的行为。编写依靠运气才可以工作的代码是一种很不好的习惯，如果我们希望删除一个数组，不管它里面元素的类型是什么，我们都应该使用 delete[]这种语法形式。

9.7.4 利用商用的泄漏检测工具

在市面上已经有好几种用来对内存使用情况进行更完整检测的产品。如果可能的话，利用它们吧；我们可以假设任何大型的、没有经过内存泄漏检测的 C/C++程序中都存在有内存泄漏。

9.8 名字冲突

如果在程序中的两个库同时定义了一个全局性质的名字，那么在链接时我们就会得到一个名字冲突的错误：

```
//在文件 Buffer.C 中

void flush_all() {
//此处忽略细节
```

```
}

//在文件 Toilet.C 中

void flush_all() {
//此处忽略细节
}
```

如果上面的两个函数被定义在还定义了其他符号的文件中，那么这些符号就（很可能）会强迫我们在运行时同时装载这两个文件的目标文件，然后装载器（loader）就会向外报告有被重复定义的符号。如果这两个函数所存在的文件没有定义其他符号，那么在运行时装载器只会装载一个文件，并调用它里面定义的 flush_all()函数。

相对 C 来说，C++在这方面仍然有了很大的提高：因为对于一个普通的（具有 C++链接属性的）函数来说，它的参数类型也是其签名的一部分，因此下面的两个名字就不存在着冲突的问题：

```
//在文件 Buffer.C 中
void flush_all(Buffer*);

//在文件 Toilet.c 中
void flush_all(Toilet*);
```

即使相对 C 来说其发生几率已大为减少，但我们仍然无法保证在两个库中不会出现名字冲突的现象，使得它们无法同时被用于同一个程序中。在某些模块化的特性被加入 C++之前，我们所能做的只是减少向全局名字空间导入新的名字，以此来将我们的类设计得尽可能少地出现名字冲突的情况。

下面我们将给出一种可行的做法。它的目的就是将类名和外部符号名混和起来，使得当两个库定义了同样的外部名时，只要它们没有定义同样的类名，这两个名字就不会冲突。

1. 对于类型名字来说，采用通用的大写形式，但对于函数和数据的名字来说，我们不要采用这种惯用的命名方式；这使得我们可以避免出现在类型名字和非类型名字间的冲突。

2. 选择不大可能与其他库相冲突的名字作为我们的类名

3. 将上面的那些名字作为我们的全局函数签名的一部分或者是全局数据，必要时使用静态成员。

9.8.1 类

选择一个不大可能与其他库相冲突的名字来作为我们的类名是很重要的一件事。我们应该避免使用类似于如下的名字：

- Object 以及从中派生出来的其他名字（如：Obj）
- Node

- Info
- Unit

如果我们构建了一个以 Object 为根的继承架构，那么当其他的库也这么做时，我们的类就很难被人所使用。

对于嵌套类来说不存在着冲突的问题，因为封装嵌套类的外层类就是它名字的一部分：

```
class Entity_network {
public:
    class Object {
        //此处忽略细节
    };
};
```

在上面的代码中，Entity_network::Object 就不会与其他任何的 Object 类相冲突。

9.8.2 函数

如同前面所说的那样，只要我们可以保证类名的惟一性，那么在参数列表中包含该类的函数就不可能与其他函数发生冲突：

```
class Telephone_number { /*...*/ };

void dial(const Telephone_number&);
```

上面的 dial(const Telephone_number&)就不会与接受其他参数的 dial 相冲突。

对于成员函数来说，解决名字冲突也很容易，因为成员函数的签名中包括了它所在的类的名字：

```
class Telephone_number {
public:
    void dial();
};
```

对于其他不是 Telephone_number 中成员的 dial 函数来说，Telephone_number::dial()不可能与它们发生冲突。

对于普通的（非成员）函数来说，如果它们不需要参数或所需参数为内建类型，那么名字冲突就有可能发生。这样的函数仍然能够被放置在全局名字空间之外。如果函数不是库接口的一部分，并且所有对它的调用都来自于同一个文件，那么最好的做法就是赋予它静态的链接属性（static linkage）（这一点和 C 中的做法一样）。否则，阻止函数进入全局名字空间的最好方式就是把它声明为某个类的静态成员函数。如果我们需要定义一些全局的数据，但又无法让它们的使用被局限在某个文件中（file static），我们仍可以通过把它们声明为某个类的静态数据成员来阻止它进入全局名字空间：

```
class Displayed_object {
```

```
private:
    static set<Displayed_object> all_objects;
public:
    static void display_all_objects();
};
```

在上面的代码中，all_objects 和 display_all_objects 都不在全局名字空间中。除非我们是在 Displayed_object 的成员函数或嵌套类中引用它们，否则我们在使用它们时都必须加上封装它们的类的名字：

```
void refresh() {
    Displayed_object::display_all_objects();
}
```

静态成员也可以是私用的；这点可以被用来阻止用户来访问我们实现细节中的非成员函数和数据。

具有C链接属性的函数

具有 C 链接属性的函数其签名中不包含它们的参数信息。因此，只要名字相同，不管它们所带的参数类型如何，所有具有 C 链接属性的函数都会发生名字冲突：

```
// 对具有C链接属性的函数的糟糕命名
extern "C" void write(int fields, Telephone_number* tn) {
// ...
}
```

在上面的代码中，它的作者试图让它可以在 C 中被调用；但该函数会和标准 C 库中的 write 函数相冲突。一旦我们让一个 C++函数具有 C 链接属性，我们就可以在 C 中调用它，同时我们还使得它可能会和其他的具有 C 链接属性的函数（所有用 C 编写的函数都包括在内）相冲突。

9.9 性能

C++成功的原因之一也“继承”自 C：C++程序的运行速度很快。这并不意味着所有的 C++程序都已经运行得足够快了。很多通常用来使得 C 程序运行得更快的技巧也适用于 C++，如：明智地选择使用寄存器变量，以及使用指针而不是数组下标。如果我们检查一段关键的代码，它里面的每行语句都已经被最大程度地优化时，我们就很容易发现这一点。

然而，随着程序员对于数据抽象（以及库的使用）的使用技巧提高，他们开始在一个较高的层次上面进行编程。这使得单个的程序员可以完成比以前要多得多的功能；但这也意味这，除非某些关键的代码段，他们也变得不再去关心每行程序间的性能问题了。相反的是，程序员将会在一个更高的层次上来关注优化这个话题。

在开始进入讲解“如何让程序运行得更快”之前，我们先给出一个警告：程序的正确性要

比它的速度更重要。和人力资源相比，计算机的运行时间的开销会变得越来越便宜；对于那些在提高性能的同时还会导致程序的可理解性和可维护性变差的方法，我们应该持谨慎对待。

9.10 不要去猜想，而应该度量！

人们在预测大程序中的性能瓶颈时总是会得到错误的结论。当使用了抽象数据类型时，由于实现细节被隐藏（这也是数据抽象的目的之一）同时也会将潜在的性能瓶颈给隐藏，我们就更容易验证这一点。即使在类的文档中已经给出了它的性能特性，但我们又怎么能保证文档没有错呢？所以说，我们应该使用性能剖析工具来实际度量我们程序的性能——而不是去猜测。通过对性能的度量，我们就可以将注意力集中在真正的性能瓶颈处，从而避免把精力浪费在优化那些只是占据运行时间1%的代码上。

当发现一个性能瓶颈时，我们首先要做的就是确认它是一个真正的性能问题而不是一个bug。如果我们在一个特定的构造函数上花费了大量的时间，而起因只是因为程序中的某个bug使得我们不得不创建了上万个这样的对象，那么提升该构造函数的速度并不是一个正确的做法！我们应该检测我们的性能剖析器所输出的函数调用次数，以及其他能够获取的信息——如每行被执行的次数。只有当数字正确时（也就是说我们的程序是按照我们所设想的方式运行时）我们才应该去考虑优化的事情。

9.11 算法

最大的性能问题（也是性能提示最戏剧化的地方）出现在我们应用程序所使用的基础算法中。在考虑寄存器变量前，我们应该回过去看看我们的算法，并对它们的复杂度进行密切的关注。对于小的测试用例来说，一个具有$O(n^3)$时间复杂度的算法不会给性能带来什么样的问题，但当n变为原来的20倍以上时，它就可能在程序的运行时间占据极大的份额。

下面我们将给出一个不佳算法的例子，假设在第8章中我们给出的List类中有一个成员operator[](int i)，它被用来存取List中第i个元素，也就是说，lst[0]就是lst中第0个元素，lst[1]是它的下一个元素，以此类推下去。假设现在我们对list进行遍历时使用了下面这种新的方式：

```
for (int i = 0; i < lst.length(); ++i)
    look_at(lst[i]);
```

对于上面的循环来说，它的复杂度是多少呢？它依赖于List::length()和List::operator[]的复杂度。这两者应该可以在文档中找到；为了简化该例子，我们假设调用length()的时间是固定的。如果该时间不固定、或者这个时间不是很短的话，我们可以改写循环如下，使得现在我们只需调用一次length()：

```
List<Thing> lst;
int len = lst.length();
```

```
for (int i = 0; i < len; ++i)
    look_at(lst[i]);
```

接下来我们假设 operator[](i)所需的时间和 i 成正比。(如果我们是从 list 的头开始寻找第 i 个元素的话，这个假设就是对的。)如果 len 就是 list 的长度，那么我们上面那个循环(它会对 list 中的某个元素调用一次 operator[])的复杂度就是 $O(n^2)$。

一旦我们得到了这样的分析结果，修正它的方法就很明显了：使用 list 的迭代器来对 list 进行步进——因为从 list 中的一个元素步进到它的下一个元素所需时间应该是 $O(1)$(常量时间)。另外一种可行的方法就是改进 List，让它能够缓存下上一次调用 operator[]的结果，然后在随后的调用中我们会从被缓存的地方开始。这样做也能够工作，但一旦有其他操作将 List 的状态改变时，我们就不得不同时也去清空该缓存。

9.11.1 将性能复杂度记录到文档中

当面对性能时，数据抽象就是一柄双刃剑。当库的实现对用户来说是可见时，用户就可以得知那些操作开销比较小，那些开销比较大。如果我们对数据的抽象实现得很好的话，那么对于用户来说实现就不再是可见的，他们也就不可能会对某个操作的开销有着很好的了解——除非我们告诉他们结果。如果我们是库的提供商，我们就应该将库中函数的性能复杂度(尤其是那些运行时间不固定的函数)记录到文档中去。这样用户就可以用这些信息来计算他们应用程序的性能复杂度。

是的，这样做可以使得我们的实现的某方面对用户来说是可见的；但我们无法避免这点。对于库来说，它的性能特性对用户来说总是“可见”的，以使得用户可以知道他们的程序需要运行多久。将我们的性能特性记录下来可以帮助我们的用户算出他们自己的程序的性能特性。

当然，当复杂度被记录下来后，它就将成为库提供者和库使用者之间契约的一部分。对某个函数的性能复杂度的修改(复杂度变高)和对某个成员函数签名的修改同样糟糕，因为它还会改变到调用该函数的用户算法的复杂度。

9.12 动态内存分配中的瓶颈

C++程序中使用动态内存分配的情况远多于 C 程序。在 C 程序中使用动态内存很容易导致程序出错，因此大家一般都尽量避免在程序中使用它；当在 C++中，使用某些在实现中使用了动态内存分配的类对象(如：String)则很简单，因为对于用户来说，动态内存的细节是不可见的。

这意味这在运行中，动态内存分配器(我们在此指的是 malloc，在其他的相同下它可能还有不同的名字)的运行时间可能会占整个程序运行时间的大部分。在本节中，我们将会讨论几种用来调整大量运行时间被耗在 malloc 上的程序运行速度的方法。

9.12.1 了解我们程序使用内存的方式

一旦确信问题不出在逻辑错误和有缺陷的算法上，我们该做的下一步就是了解内存是如何被使用的。这可以帮助我们了解如何来提炼我们的设计，或者帮助我们选择或设计一个特殊的内存分配器。我们应该问自己如下的问题：

- 有没有大量的内存被用在少数的几个类上？如果有，是哪几个？
- 我们的程序中是分配了一些固定大小的内存块，还是分配了很多的不同大小的内存块？
- 在程序运行时，它有没有释放内存，还是它在运行时持续不断地进行或多或少的内存分配，然后在程序结束时一次性地释放它们？
- 我们是否可以用一个合理的设计来减少对内存的需要？例如，如果大部分的分配都来自于复制对象，那么使用了计数器和写时复制（copy-on-write，参见第 7 章）的实现就可以帮助我们减少需要分配的内存数量。

9.12.2 尝试使用几种不同的内存分配器

目前已经有了很多不同的 malloc 实现；对于一个给定的应用程序来说，总会有一个实现要比其他的都要好。对于某些很少释放内存的应用程序来说，可能有一个最适合它的 malloc；当对于另外一个使用几种不同大小内存块的应用程序来说，可能最适合它的是另外一个 malloc 实现；对于那些需要进行不同大小内存的随机分配和释放的应用程序来说，则可能有适合它的第三种 malloc 实现。

如果 malloc 是影响性能的来源，那么我们应该用手头上能够得到的不同 malloc（或者::operator new）来构建我们的应用，并对不同的版本进行性能的度量；我们可能会对得到的结果之间的差距感到惊讶。（我曾经见过一个花费了 50%的时间在 malloc 上的应用程序因为更换了另外一个 malloc，结果那个比例变成了 5%）收集不同的 malloc 是一个不错的主意，在很多时候试用不同的 malloc 就是用来提升我们程序运行速度的最简单的做法。

9.12.3 需要时编写自己的 operator new

如果其他的都失败了的话，我们就可能需要编写自己的 operator new。如何设计一个通用的内存分配器已经超出了本书的范围。然而，我们还是可以关注这么一个常见的特例：构造对象时被分配的内存。每次我们说到“new Thing”时，缺省的意思就是：调用::operator new 来分配 sizeof(Thing)个字节的内存。这样做没有什么问题，当我们可能能够指定一个和类相关的 operator new 来更快的做到这一点（参见后面的“回顾”）。全局空间中的 new 操作符必须能够处理各种字节大小的请求，为了做到这一点，它必须使用通用的算法。然而，如果我们此时忽略掉派生类，那么一个特定的、和类相关的 new 操作符将只能接受一种大小的请求：即该类的大小。这使得构造一个使用了简单且快速算法的分配器成为了可能。

为类提供一个特殊的、和其相关的 operator new()不会对数组的分配带来影响——因为在数组分配中没有使用到那个特殊的operator new()。然而，由于对数组的分配包括一次对operator new 的调用和多次（每个元素一次）对构造函数的调用，由 new 所带来的运行时开销不大可能会在创建数组中占据重要的百分比。

回顾：特定于类（class-specific）的 new 和 delete

特定于类的 new 操作符可以通过什么和定义一个名字为 operator new 的成员函数来指定：

```
#include <stddef.h>
class Thing {
public:
    void* operator new(size_t);
    void operator delete(void*);
};
```

与全局的::operator new 相似的是，在特定于类的 operator new 中，它的第一个参数必须是 size_t（它被定义在 stddef.h 中）；函数必须返回一个 void*来指向被分配的内存。如果函数中还有其他附加的参数，那么我们只能把这个 operator new 用于 placement 语法中去。特定于类的 delete 同样也被声明为一个成员函数；它的参数是用来指向将要被释放内存的 void*。

当定义了这两个成员函数的类的对象被创建和删除时，这两个函数就会被使用。当我们创建和释放的是对象数组时，它们并不会被调用；此时我们使用的是全局的::operator new。

特定于类的 new 或者 delete 可以被派生类所继承；此时 new 的参数会自动变为派生类的正确大小：

```
class Derived_thing : public Thing {
private:
    int mem;
public:
    Derived_thing();
};
```

当 Derived_thing 被创建是，它的内存是通过如下调用所分配出来的：

```
Thing::operator new(sizeof(Derived_thing));
```

使用Pool的特定于类的operator new

在本节中，我们将会展示如何使用特定于类的 operator new 来有效地提升类的性能。在此我们使用了 USL 标准组件库中的 Pool 类；但我们在此给出的方法对于所有用来处理固定大小的内存块的高速内存分配器都是可行的。

Pool 是用来管理固定大小内存块的对象（内存块的大小是 Pool 构造函数中的一个参数）。

每次调用 Pool::alloc()都会返回一个 void*指针，它指向一块新的内存块；我们可以通过调用 Pool::free 来把这块内存归还给 Pool。Pool 的这种功能上的限制使得它可以使用比通用的 malloc 更快的算法；每次对内存的分配也就变为几条简单的指令。

通过在 Thing 中创建一个静态的 Pool 成员(用于处理大小为 sizeof(Thing)字节的内存块)，我们可以使用 Pool 来加速对 Thing 的分配。我们同时还在 Thing 中声明了特定于类的 new 和 delete 操作符：

```
//文件 Thing.h:
#include <Pool.h>
class Thing {
//此处忽略细节
private:
    static Pool pool;
pubilc:
    inline void* operator new(size_t);
    inline void  operator delete(void*);
};
```

在 new 操作符中，我们调用了 Pool::alloc 来为 Thing 分配内存。(我们将在 9.12.4 中解释代码里面的 assert。)

```
inline void* Thing::operator new(size_t bytes) {
    assert(bytes == sizeof(Thing));
    return pool.alloc();
}
```

在 delete 操作符中，我们就内存归还给 Pool：

```
inline void Thing::operator delete(void* v) {
    pool.free(v);
}
```

由于 Pool 是一个静态成员，它必须有一个惟一的定义。在它的构造函数中的参数指定了这个 Pool 是用来管理大小为 sizeof(Thing)字节的内存块的：

```
//文件 Thing.c:
#include "Thing.h"
Pool Thing::pool(sizeof(Thing));
```

当创建 Thing 时，和以往调用缺省的 operator new 不同的是，编译器会调用（内嵌的）特定于类的版本，它又会去调用 pool.alloc（同样也是个内嵌函数）来为 Thing 获取内存。这种做法对性能上的冲击将会是很明显的，我们将会在下一节中来讲述它。

在我们的List模板中添加使用Pool

我们现在来对第 8 章中的 List 做一个性能方面的根本分析。作为评测标准，我们将会使用这么一个程序，它会创建一个空的 List<int>，一个在该 List 之上的 Listiter<int>，然后使用

该迭代器来向 List 中增添 100 个整数。这个程序对于全面测试整个类的性能来说还是太简单了点，但它至少还是给了我们一些线索用来分析对链表创建和插入的性能：

```
#include <iostream.h>
#include "List.h"
main() {
    for (int loop = 0; loop < 1000; ++loop) {
        List<int> intlist;
        Listiter<int> iter(intlist);
        for (int i = 0; i < 100; ++i)
            iter.insert_after(i);
    }
}
```

当把优化打开后来运行该程序，我们发现该程序中大概有 21%的运行时间被用在 malloc 上面（这个结果来自于我的系统；对于读者的相同它的值可能会有所不同）。这看上去很令人惊讶，因为在用户的代码中并没有涉及到任何的 new 和 delete；但在 List 的实现中我们用到了它们。在特殊的情况下，向 List<T>中增添一个元素会导致一个新建 List_link<T>对象的产生。

通过改进 List_link 模板，让它使用 Pool，我们可以提升 List 的 insert（和 removal）操作的速度。我们向声明 List_link 模板的文件中增加了 Pool 的声明，以及特定于 List_link 的 new 和 delete：

```
//在List.h中:
#include <stddef.h>
#include <Pool.h>

tempalte <class T>
class List_link {
    static Pool pool;
public:
    inline void* operator new(size_t);
    void operator delete(void* v) { pool.free(v); }
    //此处忽略细节
};

template <class T>
inline void* List_link<T>::operator new(size_t size) {
    assert(size == sizeof(List_link<T>));
    return pool.alloc();
}
```

我们同时还必须在模板的定义文件中增加对 List_link<T>::pool 的定义：

```
//在List.c中
template <class T>
```

```
Pool List_link<T>::pool(sizeof(List_link<T>));
```

我们必须总是去度量性能提升上的效果——如果我们找不到其他理由，我们也可以使用“确保性能不会变得更糟”来作为我们的借口！当使用性能剖析器来分析前面给出的评测标准时，我发现在使用了新版本的程序，用以 malloc 的时间已经降至 3%，对 malloc 的调用次数也从原来的将近 100000 变为了 1196。

9.12.4 注意：派生类

为了提升我们的 List 类的性能，我们付出了一定的代价。如果有人从 List_link 中派生处新类并向里面添加了数据成员（以此改变了类的大小），那么新的派生类中继承而来的 operator new 就不能被正确使用了，因为它里面的 Pool 所管理的内存块大小和类的大小不一致。任意改变了类大小的派生类都必须声明新的特定于类的 operator new，并使用管理正确大小的内存块的 Pool。由于 List_link 是一个“私用”的类（用户甚至可能会不知道它的存在）上面的做法还是可以让人接受的。如果除我们外的其他人可能会从我们提供的类进行派生，那么我们就必须面对如下的选择：为了额外的性能，我们是否值得去对派生类做如此限制？

无论如何，安全第一，在使用了 Pool 的 operator new 中检测被创建对象的大小是否和 Pool 所管理的内存块大小相匹配是一个很好的办法。这也就是我们在前面的 List_link<T>::operator new 使用 assert 的原因。

9.13 内嵌

内嵌展开是 C++中提供的一个最具威力的性能工具。对于那些短小的函数来说，将它们展开后得到的代码可能会比调用它的代码还短——在这种情况下，选择使用内嵌展开显然要比函数调用要好。对于大多数的函数来说，哪一种做法更好并不明显。在这种情况下，内嵌展开就成了一种权衡：程序会因此变得更大（对于每个函数调用来说，它们都已经被内嵌展开为相应的代码），但也会运行得更快（此时没有存在函数调用的开销）。

当决定是否需要内嵌化一个函数时，我们的主要观点就取决于内嵌展开是否更有优势，也就是说函数的调用开销是否在函数的运行时间内占据一个很重要的部分。如果函数的调用开销在函数的典型运行时间内只占据 5%的时间，那么提供内嵌所节省的时间也只能是整个应用程序花费在该函数上时间的 5%。这样我们就不大可能得到很有效的性能提升，有时我们甚至还度量不出性能上的提升。

这就暗示着内嵌函数应该比较短小：大多数情况下它们都只有几行代码。对于一个规模大一点的函数来说，将其内嵌化所节省下的运行时间只占整个函数运行时间的很小一部分，而由此带来的空间开销则要大得多。带有循环的函数也不应该被内嵌化。迭代多次的循环所需的运行时间很容易就会将函数调用开销带来的影响给消除；而且很多的编译器也不会对带有循环的函数进行内嵌化（参见 9.13.2 节）。

9.13.1 内嵌函数调用内嵌函数

即便只是将一个只有一行代码的函数给内嵌化，我们仍然会得到某些规模有点大的函数。一个一行代码的函数可能会在其函数体中调用到两个或三个其他的内嵌函数；而这些内嵌函数里面又可能会调用到其他的内嵌函数，以此类推下去……

让我们来考虑如下这个显然应该被内嵌化的函数：

```
class Telephone_number {
public:
    // 此处忽略细节
    String area_code()  { /*...*/ }
    String exchange()   { /*...*/ }
    String number() { /*...*/ }
    String formatted()  { return "(" + area_code() +
                  ")" + exchange() + "-" + number(); }
};
```

对于 Telephone_number::formatted，它的规模会有多大呢？除了调用 area_code、 exchange 和 number 之外，它里面还有五次对 operator+(String, Sting)的调用；这些调用中的部分（或所有）都可以被内嵌化。但这还不是我们的最终分析结果——在 Telephone_number::formatted 中，还有四个临时的 String 对象被创建和摧毁（它们被用来存储函数运行中产生的中间结果）。在我的系统上面，被内嵌化的函数只包括那三个 Telephone_number 中的成员以及 String 类中的构造函数和赋值操作符，而对于 formatted 的每次调用，编译器会产生出 552 字节的目标代码！对于这么大的代码规模来说，编译器一般都会选择拒绝将其内嵌化——参见下一小节。我们再来考虑一下，如果将 String::operator+也内嵌化的话，会有什么现象发生呢？上面所得到的结果也就可以用来解释“为什么有的时候 C++程序会变得比我们预期中的还要大”这个问题。

9.13.2 被外联的内嵌

实际上 inline 指令只是一个请求，编译器也可以选择拒绝对某个特殊的函数内嵌化。这样的函数要么规模比较大，要么比较复杂；或者由于它被递归调用，使得它无法被完全的内嵌。碰到这种情况，编译器就必须为这些函数提供一个非内嵌的版本，并对调用它们的语句产生常规的函数调用代码。

很多的编译器实现会为编译单元中调用过的函数产生一个局限在文件范围内使用的静态拷贝。（这个静态拷贝也被称为被外联的内嵌。）如果我们声明了一个内嵌函数，而该函数由于其大小的原因没有被内嵌化，我们就会在每个调用了该函数的编译单元中都得到同一份被外联的内嵌。

上面产生的结果可能会带有很大的戏剧性。我曾经开发过一个带有大概 30 个源文件的程

序。因为我对性能比较关注，所以这个程序里面的大部分函数都被我内嵌化了。有一次我得到了一个早期的可工作版本，结果其大小令我很不满意。我发现这 30 个文件中的每一个都包含有 20K 字节的被外联的内嵌。而这 600K 字节已经相当于整个程序的一半大小了！

有好几种方法可以用来检测我们的程序中是否有被外联的内嵌。有时编译器会用一个标记来判断，当它产生 outline inline 时，它就会向外打印一条警告信息。另外一种做法就是运行一个类似于 nm[3]的工具来检测我们的目标文件，以此来查看定义在每个文件中的被局限在该文件中使用的函数。对于不同的系统（以及编译器）来说，情况总是不相同的，但通常我们还是可以很容易就发现程序中是否有被外联的内嵌。

一旦发现程序中存在着被外联的内嵌，我们就可以很轻松地来修正它：我们所需做的只是将那些函数改为非内嵌函数并重新编译整个程序就可以了。

9.14 Tiemann 法则

最后，所有的库开发者都应该牢记 Tiemann 法则：

“如果它从来没有被重用过，那它就不能被称为可重用的。”——Michael Tiemann

对于一个库来说，如果它没有被除开发者外的其他人所使用过，我们就不能认为它是可被重用的。这也是用来发现“库开发者是否对于库的使用做了某些（有时可能是无意识的）限制”的最佳方式。即使是新用户，他们也可能会发现库中有些功能没有提供；文档中有些部分没有被提供，有些部分则会误导使用者，甚至有的部分是不正确的；此外他们还可能会发现一些其他的依赖关系（如库只能在某个特定的编译器下工作）。

Tiemann 法则也导致了下面这个推论的产生：

“如果它不可用的话，它也就不可被重用。”

9.15 小结

- 预期的用户应该通过阅读手册中的第一行就可以判断某个库是否可以被使用。
- 在我们的文档中，类名是其中最重要的部分。我们应该审慎地选择它们，并和其他人进行交流以确保它们对其他人也有着同样的含义。
- 频繁地使用断言来即时地终止有问题的程序。
- 除非知道数组的大小上限，否则我们应该避免使用固定大小的数组。
- 为库提供一个除错版本。
- 不要使用诸如 Object 这样容易和其他库相冲突的名字来作为我们的类名。
- 使用静态成员来避免将名字导入全局名字空间。

3 译注：nm 是 UNIX 类系统中用于查看符号表的工具

- 使用对象来管理内存，不要手动去管理它们。
- 我们应该了解，进程在退出时可能不会去摧毁所有的自动对象。
- 即使里面的元素没有析构函数，我们也要用 delete[]来删除数组。
- 使用工具来查找内存泄漏的情况：编写自己的 malloc 和 free，或使用商业化的产品。
- 为了找到对已摧毁对象的使用，我们应该在析构函数中为对象中的数据成员赋予一些明显无效的值。
- 程序的正确性要比其速度更重要。
- 我们应该去度量性能，而不是猜测。
- 记录下库函数的复杂度，尤其是那些不是以常量时间运行的函数。
- 对算法进行改进通常都可以获得最大的性能提高。
- 在对性能进行调整前，首先得确定导致性能问题的原因不是程序中的 bug。
- 使用诸如 Pool 的类来构建快速的特定于类的内存分配器。
- 请牢记 Tiemann 法则。

9.16 问题

1. 我们为什么要去关心别人是否愿意在决定是否使用我们的类之前去阅读整个手册？

2. 下面代码中对于输入的大小有什么样的假设？这样的假设是否合理？

```
extern istream& operator>>(istream&, Employee&);
int total_employee_salary(istream& in) {
int total = 0;
Employee emp;
while (in >> emp)
total += emp.salary();
return total;
}
```

3. 我们在前声称，当定义了 NDEBUG 后，assert 就将被屏蔽。但如果 assert 是条件语句中的部分时，如：

```
if (kid)
    assert(kid->parent() == me);
```

当定义了 NDEBUG 后，我们是否还会去运行条件语句中的测试？如果不的话，为什么？

4. 我们可以通过将一组类嵌套到另外一个类中（其惟一的目的就是用来保证其成员的名字不会被导入到全局名字空间中去）来将它们“打包”：

```
class Telephone_project {
public:
class Telephone_number { /*...*/ };
```

```
class Telephone { /*...*/ };
class Switch    { /*...*/ };
// 没有非静态数据成员
// 没有非静态成员函数
};
```

请对这种方法的优点和缺点进行讨论。(提示：它涉及到的输入也是我们需要考虑的一个地方。)我们是否应该在 Telephone_project 中定义一些私用的成员？

5. 我们的 Arena 模板中具有这样的属性：Arena 中的内容会以一种随机的顺序被摧毁(因为我们把它们存储到一个无序的 Set 中)。请对它进行修改，使得 Arena 中的内容的被摧毁顺序和它们进入到 Arena 中的顺序正好相反。

6. 如果用户在清空其所在的 arena 前显式地删除了某个 Symbol 对象，会有什么样的情况发生呢？我们应该如何改进 Arena 模板，以使得这种显式删除元素的行为也可以被它所支持？(提示：难点在于确保不会有对象被删除两次。)

7. 当申请空间的大小和 Pool 中的设置不匹配时，一个使用了 Pool 的特定于类的 new 可能会试图去调用全局的 new 来进行内存分配：

```
template <class T>
inline void* List_link<T>::operator new(size_t size) {
if (size == sizeof(List_link<T>))
    return pool.alloc();
return ::operator new(size);
}
```

但上面的做法还是不够。为什么？(提示：考虑一下 operator delete。)请设计一个 operator delete，使得它可以和这个 operator new 配套使用，并说出我们为此所增加的处理和复杂度是否值得？

8. 在 9.7.2 中我们的 operator delete 和 print_allocated 函数中使用的是 stdio，而不是 iostream，为什么？

9. 改进我们的 print_allocated 函数(必要时也包括 new 和 delete 操作符)，使得它可以打印出当前所分配的每块内存的大小。

10. 当使用全局的 operator new 来实现时，我们可以怎样使用 Pool？(感谢 Andrew Koenig 提供的这个想法。)

Tiemann 法则是否预示着开发过程的失败？通过设计一个在开始阶段就可以被重用的库，采用一个好一点的开发过程是否可以让我们“第一次就把事情做对”？这样的过程的工作情况又是怎样的？

第 10 章

异　常

程序在运行过程中，有时候会产生错误。异常特性（见如下的评论）使得程序能处理那些非预期的情况而无需程序员显式地检查每一步代码来发现这些情况。在这一章中，我们将讨论在设计的什么地方应该使用异常，而在什么地方却需要避免。

10.1　一个负面的声明

我们将首先考察异常的基础，但是需要提醒读者的是异常是一种新的语言特性，并且在未来的数年中异常的使用方式都毫无疑问的会处于不断的演变过程之中。尤其是在这一章完成的时候（1992 年 12 月），支持异常的编译器才刚刚出现。因而我们在此建议所有的开发者都应该跟上这个领域的持续发展过程。

回顾：异常

关键字 throw 可以用来抛出一个异常，它可以使用任何类型的对象作为其惟一的操作数：

```
class Xcpt {
    //此处忽略细节
public:
    Xcpt(const char*);
    const char* diagnostic();
};

void allocate_resource() {
    if (allocation() == FAIL)
        throw Xcpt("Allocation fails!");
}
```

当 throw 语句执行的时候，控制权将被交给最近的那些可以处理这些异常的异常处理器。而 throw 语句的操作数拷贝（在上面例子中是 Xcpt）将被传递给这个异常处理器。

异常处理器通常如下表示：

```
void allocate_and_results() {
    try {
        allocate_resource();
    }
    catch (Xcpt& caught_Xcpt) {
        cout << caught_Xcpt.diagnostic() << "\n";
    }
}
```

在 try 之后的括号中的代码是一个 try 语句块，而在 catch 之后的括号中的代码就是一个异常处理器。

当程序运行到 try 语句块的结尾时，如果没有异常被抛出，那么异常处理器中的代码将被忽略，程序将接着从异常处理器之后的代码开始执行。如果一个异常被抛出，运行时系统将开始对函数调用堆栈进行展开（适当时还会调用某些对象的析构函数）来寻找最近的 try 语句块。当最近的 try 语句块被找到时，throw 关键字的操作对象将按顺序的和相应的 catch 语句中的异常对象的进行类型匹配，如果有一个 catch 语句中的对象和其类型匹配成功，那么程序控制权将转移到这个相应的异常处理器中。当这个处理器代码执行完的时候，控制权被转移到处理器后的第一条语句。

当抛出的对象和 catch 语句的对象是同一种类型，或是其派生类，或者抛出的对象是一个指向派生类的指针而 catch 语句的对象是指向该派生基类的指针的时候，类型匹配都将成功。这里并没有发生隐式类型转换。

如果找不到和其相匹配的 catch 语句，那么运行时系统将继续进行堆栈展开，以寻求其他的和其类型相匹配的 catch 语句对象。如果没有找到任何匹配的 catch 语句对象，那么 terminate() 函数将被调用，缺省情况下它将终止程序。

下面是一个可以捕获任何异常的函数：

```
main() {
    try {
        // 我们的整个程序
    }
    catch (...) {
        // 所有未被捕获的异常在此被处理
    }
}
```

一条不带任何参数的 throw 语句会将当前的异常重新向外抛出：

```
try {
    // 一些内容
}
catch (Xcpt& ex) {
    if (can_handler(ex))
        handle(ex);
```

```
    else
        throw;  // 等同于"throw ex"
}
```

10.2 为什么需要异常？

为什么我们要在C++中包含一个针对异常的特殊的语言特性呢？依照传统观点，C程序员有四种处理异常情况的方法：

忽略它们——如果程序真的出现了错误的话。对于一小部分人来说，这种快速而丑陋的方法确实是可以正确的处理这些情况。然而，用这种方法将很难构建出高质量的软件产品。

当检测到异常时直接退出程序而不提供恢复机制。对于某些错误情况来说，这样做的效果很不错；当对于大部分的应用程序来说，这种做法是不可接受的。例如，在一个程序中，我们向外界申请了一些资源（如对数据库进行加锁），如果我们不在程序退出前把这些资源释放掉，那将会导致一场灾难的产生！

设置全局错误代码或者让函数返回错误代码——这是一个可行的方法，但是细心的程序员最终将不得不花费大量的时间来检查错误代码，而大部分的错误都来自于程序员忘记检查错误代码。

使用setjmp和longjmp——当异常发生的时候，C程序员可以使用longjmp函数来把程序的控制权交给某些恢复动作。

有很多理由都表明了为什么这些处理方式对于C++来说都是不可行的。第一，在某些环境中，我们得不到用来检测的返回值。尤其是，如果在构造函数中发生错误，我们该如何应对呢？如果没有异常，惟一的选择就是终止程序或者产生一个处于某种错误状态下的对象，并且指望对象的使用者会来检测对象的创建是否成功。

C++程序不应该使用setjmp和longjmp。这是因为longjmp在对堆栈进行展开时不会去销毁处于栈中的自动对象。遗弃对象却不调用对象自身的析构函数只会给程序带来混乱。

异常针对这些问题提供了一个好的解决方法，它们同样使得程序员在编码的时候无需考虑在函数的调用过程中会发生的错误。

10.3 一个异常的例子

在9.2.1中，我们定义了一个名称为check的函数，它使用一个整型数作为其参数并且当参数为零的时候打印一条错误消息。

```
inline void check(int expr, const String& diag) {
    if (expr == 0) {
        cerr << "Interal error: " << diag << endl;
```

```
        abort();    //或抛出异常
    }
}
```

我们使用这个函数来代替经常被用来检查错误的断言——即使宏 NDEBUG 并没有被定义。现在我们重新改写这个函数，让其在检查失败的时候能够抛出一个异常。而抛出的对象将保存一条诊断消息：

```
class Check_failure {
private:
    String diagnostic_d;
public:
    Check_failure(const String& s) : diagnostic_d(s) {}
    String diagnostic() const { return diagnostic_d; }
};
```

这个 check 函数将抛出一个 Check_failure 对象而不是中断程序。

```
inline void chekc(int expr, const String& s) {
    if (expr == 0)
        throw Check_failure(s);
}
```

对于 check 函数的调用并没有改变：

```
void f() {
    char* space = new char[BUFFSIZE];
    check(space != 0, "Memory allocation");
    // ...
}
```

当 space 等于 0 的时候，check 函数将抛出一个 Check_failure 对象，而这个异常对象将被任何拥有适当的异常处理器的函数所捕获。

```
main() {
    try {
        // 整个程序
    }
    catch (Check_failure diag) {
        cerr << "Internal error: " << diag.diagnostic() << endl;
        return 1;
    }
    return 0;
}
```

程序控制权将交给异常处理器，并且那些从堆栈里弹出的对象将被正确地销毁。

10.4 异常只应该用来表述异常情况

现在，我们已经看见了一个正确地使用异常的例子。让我们再讨论一下，在什么时候应该避免使用异常。异常应该只被用来处理异常的状况；它们不应该被用于程序的正常控制流。举例来说，一个异常不应该成为你是否到达一个链表结尾的标志信号。

```
void process(const Thing&);

void process_each(List<Thing>& list) {
    try {
        for(;;) {
            Thing t = list.get();
            process(t);
        }
    }
    catch (List::empty) {
    }
}
```

上面这段代码依赖以如下事实：如果针对一个空链表取对象，List 类将抛出一个异常。这段代码的结果是正确的，但是基于某些原因，它却是一个糟糕的设计。

上面的程序流程并非显而易见（如果你被要求维护一段使用了 longjmp 来从一个循环中退出的代码，你会有什么感觉？）。任何阅读这段代码的人都需要明白控制权将交给 List::get 中的某个可以捕获异常的地方。在只需要返回一个值就可以很好地工作的情况下，为什么还要使用到异常这样的高级特性呢？

```
void process(const Thing&);

void process_each(List<Thing>& list) {
    Thing t;
    while(list.get(t))
        process(t);
}
```

上面这个版本就简单明了多了。

在伴随着异常被抛出后的栈展开过程中，任何栈上的自动对象都将被销毁。然而，这却无法保证所有的资源都会被释放，举例来说：

```
char* temp_stuff = new char[100];
use_stuff(temp_stuff);
delete[] temp_stuff;
```

如果在 use_stuff 过程中任何异常被抛出，并且在外部域被捕捉，那么程序就会发生内存

泄漏，因为 delete 将永远都不会被执行。如果这样的异常只是偶然现象，在实践中这样的内存泄漏可能不是大问题。但是如果这样的异常是作为普通事件的一部分（比如说是一个执行几千次的循环中的事件），那么内存泄漏情况就将变的非常严峻了。

为了对付这样的情况，我们可以在每一个 new 操作后面都跟随一个可以捕捉任何异常的 try 块，并且在异常处理器中，首先调用 delete，然后再重新抛出异常。

```
char* temp_stuff = new char[100];
try {
    use_stuff(temp_stuff);
}
catch (...) {
    delete[] temp_stuff;
    throw;
}
delete[] temp_stuff;
```

这样的代码很难看，也容易出错，同时更难以被测试。一个更好的办法是使用自动对象来管理资源（就像我们在 9.3.1 中处理内存的那样）。然而，即使我们是这么做的，我们所使用的其他函数库会不会也这样做我们就不清楚了。

除非我们非常细心，否则用于正常程序流的异常将导致异常的产生。回到前面的循环，假设 process(const Thing&)使用了一个临时 List，并且包含了一个“当 List 为空时自身是 List 的一个 get 操作”的逻辑错误。在那个例子中，控制流将忽略 process_each 中的异常处理器过程，并且没有认识到，那并不是一个一般异常。

存在一些不同的异常实现机制，它们都使用了一些不同的运行时系统特性。异常的执行通常总是比一般的直接处理程序要慢。假设你刚好拥有一个对异常采用快速实现的系统，一个随之而来的隐藏含义就是当你的代码被移植到一个对异常采用慢速实现系统中的时候，代码的执行性能将受到冲击。

10.5 理解异常

考虑一下，如果我们要为异常事件保留异常的话，我们该怎么办？当在为对付特殊的异常状况伤神的时候请对自己问如下两个问题。

是谁或者什么导致了这个问题的产生？

谁或者什么将处理它？哪个部分将负责检测它们？尤其是，程序是否企图从错误中恢复过来，或者只是或多或少的给出一个比较优雅的诊断消息后就简单的退出？

第一个问题的答案通常也决定了第二个问题的答案。

10.6 责任评估

首先，我们要搞清楚是谁或者什么导致了这个问题。最有可能的候选者如下：

10.6.1 你

属于这种情况的是当你的程序检测出了一个内部逻辑错误(比如说一个断言失败)的时候。其他的例子是那些不会发生的情况，比如在不该有的地方发现了一个空指针。

大多数程序都不会尝试去分析内部的逻辑错误。它们将会产生一个可读的诊断(例如，"Internal error number 234; contact support organization")，然后或者是终止程序或者是执行一些高层次的恢复动作。

10.6.2 其他的程序员

如果你是一个库提供者，你将在你所提供的函数中的输入操作中提出一些限制。而必须遵守这些前置条件的责任就落到了使用你的库的程序员身上。如果在前置条件被破坏的情况下函数被调用，那么就应该产生一个相关的异常。比如说企图从3个组成的字符串中取第5个字符或者给一个函数传递一个非预期的空指针。

再次声明，用户的程序也不一定会去分析这些错误。出错消息应该能够让其他程序员理解，但是大多数程序都不会关心这两种情况之间的区别。

10.6.3 资源的提供者

有些问题并非是逻辑错误，而是由于在运行时对于资源的获得失败造成的。这些错误可能包括运行时内存耗尽，取得数据库锁失败，向磁盘写文件时磁盘满，系统调用超时，或者对于一些其他的资源申请失败（如许可证管理器中的许可证过期）。

这些错误是程序最有可能进行恢复的。举例来说，一个内存耗尽的程序也许可以以其他的方式从一些地方获得内存资源，取得数据库锁的失败可以通过等待一段时间后再去对数据库进行取锁的尝试来解决，从许可证服务器获得许可证的失败可能通过给出一个诊断并且提供给用户一个选择——等待许可证重新可用或者退出程序或者预定一个新的许可证等等来进行处理。

10.6.4 用户

这些错误是由一些用户所导致的。它们可能包括破坏了输入的限制（比如在要求输入数字的时输入了一个字母字符）或者请求了一个不合适的操作（对一个只读文件进行写操作）。

对于用户的错误我们不应该使用异常。异常的最大好处就是可以让处理用户错误的代码和发现这些错误的代码相分离。然而，发现用户错误的代码往往也是处理这些错误的最好的代码。同样的，当我们希望从任何序列的操作中捕捉错误以及以相同的方式进行处理的时候，异常也

是非常有用的。但用户总是要比程序员更难以对付，而我们却想独立地处理每个用户的错误。如果我们想让每一个用户的错误都拥有一种特殊的处理方法，那么那种检查每一个操作的结果的老式方法也许是合理的。

当然，在计算机科学中没有绝对。用户可以在我们程序中的许多地方描叙文件名，而我们希望在最高层次上像下面的方法那样捕捉到任何错误的文件名：

```
main() {
    try {
        // 整个程序
    }
    catch (Bad_file_name n) {
        cerr << "Bad file name: " << n << endl;
    }
}
```

但是这样的代码对于一个严肃的商业化产品来说肯定是不够好的。

10.7 设计异常对象

在进行异常对象的设计之前必须理解谁（或者是什么）来检测这个异常。如果是人负责检测这个异常，那么异常对象就应该本质上只是字符串。程序通常只关心异常的发生，而不是问题的细节。如果是由程序负责检测异常，异常对象就可能变得复杂一些，因为它将包含一些程序在检测中需要用到的数据。

举个例子，一个数学库是否应该针对除0操作和溢出提供独立的异常？

```
// 过于复杂的异常结构
class Divide_by_zero_error {
// ...
};
class Overflow_error {
// ...
};

try {
    call_math_function(x,y);
}
catch (Divide_by_zero_error) {
    // ...
}
catch (Overflow_error) {
    // ...
}
```

除了要指出需要打印哪种诊断，很难想像针对除0和溢出这样的异常程序还会有什么不同

的响应。大多数程序只是在错误发生的时候被通知，所以它们可以打印出诊断消息给用并且退出（或者以其他的某种方式恢复）。所以使用一个单独的异常类就足够用来处理这样的情况：

```
class Math_error {
// ...
public:
    String diagnostic();
};

// ...

try {
    call_math_function(x,y);
}
catch (Math_error e) {
    cerr << "Math error: " << e.diagnostic() << endl;
}
```

但是，我们也许会问，为什么不为那些需要关注的程序提供不同的异常呢？它的代价是什么？每个异常，就像其他的特性一样，都会给程序增加复杂性，而复杂性就是一种潜在的代价。异常必须被记录为文档，而这将使查找类中重要部分变得更加困难。即使是在异常可以被安全地忽略的情况下，程序的每一个使用者也都必须理解异常。

回顾在第一章曾经讨论过的：在不确定情况下，我们偏向于把接口做得尽可能的小。任何遗漏的缺陷都可以通过添加新的异常类来弥补。但是如果你一开始就提供了大量的不同异常类，即使它们大多数都永远不会被用到，你也将受困于那些文档以及维护它们的工作。

10.7.1 异常和继承

当设计异常对象的时候，我们可以考虑通过继承来封装两个或者多个类所持有的共同概念。比如，我们可能会想拥有一个针对包含着可读的字符串异常的抽象基类 Diagnostic：

```
class Diagnostic {
public:
    virtual String diagnostic() = 0;
    virtual ~Diagnostic();
};
```

我们的 Check_failure 和 Math_error 对象和这个抽象是一致的，所以它们可以派生自 Diagnostic：

```
class Check_failure : public Diagnostic {
private:
    String diagnostic_d;
public:
    Check_failure(const String& s) : diagnostic_d(s) {}
```

```
    virtual String diagnostic() { return diagnostic_d; }
};

class Math_error : public Diagnostic {
//此处忽略细节
};
```

调用者可以通过使用一个包含 Diagnostic&匹配类型的处理器来捕捉任何从 Diagnostic 继承下来的对象。

```
main() {
    try {
        // ...整个程序
    }
    catch (Diagnostic& diag) {
        cerr << "Internal error: " << diag.diagnostic() << endl;
        return 1;
    }
    return 0;
}
```

注意，调用者必须捕获针对基类的引用。捕获工作就像是函数参数的初始化一样；如果捕获对象类型被描叙为 Diagnostic 类，那只有属于 Diagnostic 对象的部分会被捕捉（对象会被切割）。

针对我们的继承体系的一个合理细化就是提供一个 Src_diagnostic 新类，它也是一个同时包含了字符源和行数信息的 Diagnostic：

```
class Src_diagnostic : public Diagnostic {
public:
    virtual String source() = 0;
    virtual int line_number() = 0;
};
```

这个异常可以同时被人或者程序检测。当 Src_diagnostic 被抛出的时候，一个程序环境也许会使用 source 和 line_number 把窗口中心提升到指定的行数。另一方面，一个 Src_diagnostic 就是一个 Diagnostic，一个针对 Diagnostic&的捕捉同样可以成功的捕捉 Src_diagnostic 并且打印出它的诊断字符串。

避免使用Exception作为类名

不要把你的任何类命名为 Exception。这样的名称很可能带来名字冲突：这就使得其他的库不太可能使用相同的名字。而其必须负担的更高风险代价是，ISO/ANSI C++标准委员会正在考虑创建一个标准的 Exception 类[1]。

1 译注：标准库中的异常类名字叫做 exception，处于 std 名字空间里面。

10.8 小结

- 异常是 C++中的一个新的特性，它正在不断变化中；请时刻关注标准化组织在这个领域方面的进展。
- 不要将异常用于正常的程序控制流程中。
- 在理解了异常将被人或者是程序来检测的基础上再设计你的异常类。
- 在你的异常类的相关联设计中使用继承。
- 不要把给类取名为 Exception。

10.9 问题

1. 在哪种情况下，程序可以使用 setjmp 和 longjmp 来同时处理 C 和 C++的代码？

2. 在 Ada 中定义了如下的标准异常：

- 约束错误——某些操作会破坏输入的限制（如：对数组的越界索引）。
- 数值错误——对于数值的操作产生的越界结果（如，上限溢出和下限溢出）。
- 程序错误——检测到的程序逻辑错误（bug）。
- 存储错误——对内存的申请失败。
- 任务错误——在两项任务的通信中出现的错误。
- 时间错误——到达某些操作的时间限制（time-out）。
- 输入/输出异常——这一类型的异常包括各种各样的输入/输出错误。

针对上面的异常情况，请判断哪些适合与人交互，哪些适合和程序交互，如果使用的是 C++，我们应该如何处理它们？

3. 编写一个 Array 类，当它碰到越界引用时会向外抛出一个异常。我们应该抛出什么类型的对象呢？

4. 考虑下面的对于 Diagnostic 和 Src_diagnostic 类的另一种实现：

```
class Diagnostic {
private:
    String diagnostic_d;
public:
    Diagnostic(const String& s) : diagnostic_d(s) {}
    String diagnostic() const { return diagnostic_d; }
    virtual ~Diagnostic() {}
};

class Src_diagnostic : public Diagnostic {
private:
    String  source_d;
```

```
    int     line_number_d;
public:
    Src_diagnostic(const String& diag, const String& src, int line)
    : Diagnostic(diag), source_d(src), line_number_d(line) {}
    String source() const { return source_d; }
    int line_number() const { return line_number_d; }
};
```

请将它们和原有的版本进行比较，说出它们之间的权衡。

5. 请对 Src_diagnostic 进行改进，让 diagnostic()成员函数返回的值包括异常发生时程序所在的文件名和代码行数信息。

第 11 章

向 C++移植

将一个项目移植到 C++上去所需要的变动不仅仅是对源代码的修改。它还包括对项目组成员的培训，大家适应 C++所需要的时间，新的设计范型，新的开发工具，以及新的开发过程。对于使用 C++和面向对象来进行程序开发的过程的完整讨论已经超出了本书的范围；但是，我们还是试图在本章中对那些经常出现的一些问题进行简单的讨论。

本章并不讲述“如何管理 C++项目”。相反，我们试图为多人参与的项目组成员（而不仅仅是老板）给出一些关于“如何构建一个真实的 C++程序”方面的提示。

11.1 选择使用 C++

首先，项目组中最重要的一个决定就是决定在项目中采用 C++进行开发。大量的 C++编译器、工具、函数库以及允许环境使得我们比以前更容易做出选择使用 C++的决定，不过我们仍然还得面对伴随着利益而同时出现的风险。

11.1.1 学习曲线

最显而易见的问题就出现在对 C++的学习过程中。当向 C++移植时，存在着两个完全不同的学习曲线：对语言的语法和语义方面的学习；以及对面向对象设计的学习。

C++中包含有大量的语言特性，但我们并不需要完全掌握它们也能高效地使用这种语言。例如，同我交谈过的大部分程序员都从未使用过指向成员的指针。这并不是说这种语言特性没有用，只是只有很少的一部分应用程序才会需要使用它们。（这也是我们为什么没有在本书中讲述指向成员的指针的原因。）大部分熟练的 C 程序员都能够在一周内学习完那些基本的语法特性——类、成员函数、单继承……这并不会使得他们同时成为 C++方面的专家；但他们还是可以在同时学习 C++的其他特性时很好地利用已经学会的东西来构建有用的程序。

学习面向对象设计会需要我们更多的时间。对于熟练的 C 设计者来说，大概需要 3 到 6 个月的时间给他们来进行这种基本观念上的转变。但这也不意味着开发人员在这段过程中就不能做出有用的设计，或者他们的第一个面向对象的设计模型一无是处；相反，这段时间只是用来让设计者习惯于用面向对象的方式来表述他们的设计。

项目的开发进度表中应该为这种学习曲线留出足够的时间。我们并不能找到一种“艰苦且快速”（hard-and-fast）的规则来计算出这段时间的长度；不过 Jossman 等人还是给出了一些数据：在经过一年“黑暗中摸索”的日子，他们的项目从 C++中所获得到的生产力的提高所带来的收益要远大于他们在前面那段时间为此付出的学习时间。从另一方面来说，一个采用 C 开发的、具有一个三个月后必须交货的最终期限的项目是很难从 C++中获得到什么帮助的。

11.1.2 语言及工具的完备

C++已经发展到了一定的阶段，此后对于它的语言方面的改进将变得十分缓慢。众所周知，C++是一门规模庞大的语言，向它里面再新增额外的语言特性势必将使得它更难以被掌握。除此之外，对大量的已有代码的兼容性方面的限制也使得语言的改进变得困难起来。ISO/ANSI C++标准委员会希望能够在 20 世纪 90 年代中期发布出一份官方的 C++标准文档[1]。当新的标准被制订出来后，所有主要的 C++编译器的提供厂商都将使得他们的编译器遵循这份文档。

然而，有关 C++的工具、函数库以及开发环境则并没有达到同样的成熟度。每年我们都可以看到大量的新的 C++开发工具的出现。将设计和实现阶段完全地整合起来的面向对象设计的工具也才刚刚出现。

11.1.3 应用程序的限制

“C++不是最好的 Smalltalk；Smalltalk 才是最好的 Smalltalk。”——Bjarne Stroustrup。

C++是一门优秀的通用编程语言；但这并不意味着对于所有类型的应用程序来说它都是最佳的选择。在 AI 领域中，LISP 仍将是第一选择；在程序的可扩展性和动态更新比程序的运行性能以及静态类型检测更重要的应用程序中，Smalltalk 可能更适合一些；在构建一些小的、性能不是最主要考虑的程序时，程序员可能会采用 AWK 或者 Perl。虽说 C++也可以做到这些，但这并不意味着我们可以就此忽略掉所有其他的编程语言。

11.2 采用 C++

对于开发组中学习 C++的情况，存在着两种意见分歧。一种就是：程序员都很懒惰，在没有被强制要求的前提下，他们是不会取主动学习面向对象的设计以及其他 C++的高级特性的。持这种观点的人认为：在进行 C++项目的开发时，我们必须在一开始就把事情给做对，设计好一个完美的面向对象的框架，然后从开发的第一天开始就使用对象来处理每一件事情。

经过好几年对于采用 C++的项目的观察，我个人更能接受另外一种观点，它认为项目组中可以通过如下三个阶段来更“容易”地使用 C++：

- 一个更好的 C；
- 用 C++进行数据抽象；

1 译注：第一份 C++官方标准文档（ISO/ANSI 14882）在 1998 年被发布出来。

- 面向对象的编程方式。

11.2.1 将 C++作为一种更好的 C

在开始的阶段，我们应该将 C++中相对 C 的新特性作为一些对 C 的扩展来使用。我们并不需要取改变我们的设计方法和编程范型；但我们仍可使用诸如参数类型检测、函数重载、内嵌函数、常量等这些 C++中的新特性。C++中提供的诸如 String 这样的已有类也可以在我们的项目中被使用。在这个阶段，我们能够得到的好处就是整个项目的 C 程序员（无论其技巧和经验如何）都可以在短时间内使用 C++来完成一些有用的工作。

11.2.2 数据抽象

下一个阶段我们就要开始设计和实现类了，在这个阶段中我们的关注开始转向将接口和实现分离。我们在这个阶段中设计的类大多是用来封装一些应用领域中的基本概念，设计和实现它们的人也通常都是项目组中最具经验，技术最好的成员。此时我们还很少涉及到继承和虚函数这些特性。

11.2.3 面向对象编程

在最后的那个阶段中我们开始采用面向对象的设计方法。此时继承和虚函数开始成为项目中不可分割的部分了。由于我们很难将一个已有的设计转换为一个面向对象的设计，所以我们通常都是在一个新的模块中或者对一个已有模块进行重新构造的过程中采用面向对象的设计方法。

11.2.4 三步和一步

对于上面的分三步走的批评指出：程序员以及整个项目都容易停留在开始的阶段中。但据我所见过的项目来说，这种情况并没有出现。当项目组成员开始熟悉数据抽象和面向对象的编程方法之后，使用它们也就显得极为自然了。

另外一种批评持有的论点是：第一个 C++的设计将带有严重的 C 的风格，而它们本来可以在面向对象的形式下完成得更好的。但我们必须考虑到，设计它的人还只是一个在 C++和面向对象设计方面的初学者。C++中提供了很多的素材，然而同时在项目中使用这些素材将给项目带来极大的风险。面向对象设计对于大家来说是一种思维方式的基本转换。而让一个有着 40 个人左右的项目组中的成员（忽略掉他们之间的经验和技能）同时进行这种转换也存在着极大的风险。

11.3 设计和实现

在本节中，我们将讨论一些在项目移植中的详细设计和实现阶段的前期设计中可能会碰到

的问题：

11.3.1 使用 C++会导致设计阶段中的前期负担变重

C++中鼓励设计者在编码前花费大量的时间来充分地理解他所做的设计。这样做很有好处，因为在设计阶段中发现的问题较容易被解决；但这样做也带来另外一个问题，那就是：在我们进行第一行代码的编写前，我们必须花费大量的时间。

11.3.2 迭代的设计方式

软件设计是一个迭代的过程。在一开始，我们进行了一个最初的设计，得到了一些初始的类，并且实现了一些较高层次上的函数。此时，类的设计者通常都会发现，他们设计的那些初始的类中某些重要的场合下都会有问题。使用类的经验也告诉设计者，他们对设计的理解还不够充分。但这并不意味着设计者在编码前没有对设计进行仔细的思考。在一开始就要求人们能够预期到所有可能出现的问题及其重要性是很困难的。设计时出现的问题（设计是在一个想像的空间中进行的，试图设计出涵盖所有代码功能的模型）经常使得设计者花费大量的设计时间在那些程序中不那么重要的部分上。

使用数据抽象可以帮助我们更好地进行迭代，减小因为特殊的改动而导致整个程序推翻重来的可能性。即便我们做的改动很重要，只要最初的类的设计很合理，它们的实现依赖细节被很好地隐藏，我们就可以使用 C++来很好地改进这个设计。而一个面对如此程度改动的 C 项目则将不得不从头开始或者（迫于时间压力）继续采用原有的设计并期望事情能够按照最好的方式发展。

11.3.3 专家们来设计类，其他人只管使用它们

构建 C++类要难于使用它们；项目中的基础类应该是由那些技能最精、经验最丰富的人来设计的——数据抽象并不能去除掉诸如存储管理、簿记以及设计实现这些内容。这样做的好处就是：只有类的设计者才需要去关心这些内容；而它们的使用者则不需要。类的设计者必须保证这些类的正确性以保障项目的成功。一旦基础类是正确的，那么项目组中的其他成员也就可以不需要导入其他的簿记工作来使用它们了。

11.3.4 不要试图同时使用所有的特性

C++中有着许多的特性；但不是每个应用程序都会使用到所有的这些特性。某些 C++新手可能会因为“既然 C++提供了它们”而试图使用所有的新特性。即便试验是学习过程中的重要组成部分，我们还是应该牢记住产品和试验之间的不同。采用这些新特性总是很容易的，但这却会容易导致我们得到的是一种具有“巴洛克”风格的设计。多重继承、隐式类型转换以及操作符重载是这些特性中最容易产生问题的。每个程序员都应该能够解释为什么在他的程序中使用某种特性，这种特性是怎么使得他们的程序运行得更快、更安全，而且更容易被理解的。

11.3.5 使用通用的库

使用 C++带来的一个最大的好处就是它可能允许程序员在高层次上的抽象进行操作。当我们在使用一个包含有模板容器类的库时，在详细设计中用于构建数据块的就不再是数值、结构以及指针了：它们现在变成了链表、集合、bag、map 以及 block。这使得设计的实现者可以从重新实现这些数据结构的痛苦（这种重新实现通常也容易导致 bug 的产生）中解脱出来。当库被大量使用时，我们的编码速度就可以得到明显地提高；程序员可以在不关心 List 和 String 的实现细节的前提下使用 List<String>。

11.3.6 分配更多的编译时间

C++允许程序员在相对于 C 来说更高的层次上进行编程，尤其是在库被使用了后这种区别更为明显。然而，即使程序员不再需要去关心低层次的细节，编译器和链接器仍然需要去关注它们。一条语句做的事情越多，编译器所化费在它上面的编译周期也越多。当模板被使用时这种情况就显得更明显了。

与人所花费的时间相比，编译时间要廉价得多，所以上面的做法是一种很好的权衡；但我们不应该忘掉，因为我们做了这么样的权衡，那么在代码中导入的更多的自动化操作将会导致更多的编译时间被花费。

11.3.7 目标代码尺寸的增大

从 C++源代码所产生的目标代码可能会非常大。这并不是语言的问题——将两个整数进行相加这个操作在 C 和 C++中产生的代码是一样的。这主要是因为在 C++中，一条语句能够做的事情要更多了，对内嵌展开的大量使用使得这种情况变得更为明显。

C++使得库的使用变得更为简单；但库也同样会导致目标代码变大。我们可以预料到，一个用于通用性质的库其目标代码尺寸总要大于我们对它们里面某些功能进行重新实现所得到的目标代码尺寸。将库分割成不同的目标文件可以减轻这种影响，但并不能消除它。

11.3.8 将现有的 C 代码转换为 C++代码

通常，我们都不用去担心此事，尤其是当 C 代码并不是当前被开发的代码时。C++和 C 在链接时是相兼容的，程序员也似乎可以同时维护这两种语言所编写的代码。我们可能不得不编写 C++的头文件来声明 C 函数，但除非 C 里面的代码出现了问题，否则通常是不需要去修改它们的。我们可以使用 Makefile 来对不同的源代码文件选择使用哪种编译器。

11.4 开发一个坚实的基础

我们很容易得到通用的库；但我们可能很难得到那些封装了我们正在构建的应用程序所属

领域中重要概念的库。正在开发用于操作同一应用领域产品的组织可能会发现，为他们的产品开发一个自有的可被重用的库是一件合算的事情。一旦这些库被正确地完成了，他们就可以节省大量的费用；但我们通常都会低估开发一个可重用的库所需的代价。

重用不能只是一种业余爱好。我们不能期望从正在构建的应用程序代码中同时得到可重用的类。构建一个可重用的类所需花费的时间可能是构建一个足以在应用程序中工作良好的类所需时间的好几倍；可重用的类必须更具通用性，更健壮，被测试得更充分，记录得更具体，以及支持得更全面。所有的这些额外的特性都必须明确地在它们中得到体现。

将开发通用的库作为一件明确的行为同时也会帮助我们确保不同的库使用一致的方式来进行使用，如果在我们的库中有一半的代码使用某种遍历方式，另外一半则使用另外一种遍历方式，那么它就很容易让用户产生混乱。

11.5 相关的思考

使用 C++并不会导致设计者和程序员的经验、体会以及判断变得可有可无。相反的是，它给以了设计者和程序员工具以帮助他们在更高的概念层次上应用他们的技能。差的程序员在使用 C++编写程序时同样也会产生差的程序；但好的程序员则可以使用 C++所提供的特性来快速地构建快速的、易于理解的以及易于维护的程序。

11.6 小结

- 项目进度表必须为学习曲线预留余地。
- C++是一门优秀的通用编程语言，但对于特殊的应用程序来说，存在着更适用的编程语言来实现它们。
- 请逐步地采用 C++，一开始将它作为更好的 C 来使用，然后开始采用数据抽象，最后使用面向对象的编程方式。
- 在使用 C++的设计过程中，前期将会变得负担更重。
- 不要同时使用所有的特性。
- 设计应该采用迭代的方式来进行。
- 使用类要比构建它们容易。
- 尽量使用通用库。
- 用于处理高层次抽象的语言需要更多的编译周期。
- 我们程序的目标代码会变得更大。
- 除非我们计划对其进行新的开发，否则不要将已有的 C 代码转换为 C++代码。
- 重用不是一种业余爱好。

11.7 问题

1. 项目的大小是如何影响我们采用 C++的决心的？它是否会对我们是应该直接采用面向对象的编程方式还是逐步进行这两种判断产生影响？

2. 对于目标文件大小有某些限制的项目，我们应该怎么样把它移植到 C++中去？我们会采用什么样的过程来控制目标代码的大小？

3. 那些不具有丰富经验的程序员是否可以构建出用于创建优秀程序的类？这样做会不会导致项目的失败？我们该如何做才能让那些不是很优秀的程序员也能编写类？

4. 我们应该如何跟踪设计中的迭代，以确保我们能够作出正确的修正？

11.7 问题

1. 项目的大小是如何影响我们采用 C++的决心的？它是否会对我们是应该直接采用面向对象的编程方式还是逐步进行这两种判断产生影响？

2. 对于目标文件大小有某些限制的项目，我们应该怎么样把它移植到 C++中去？我们会采用什么样的过程来控制目标代码的大小？

3. 那些不具有丰富经验的程序员是否可以构建出用于创建优秀程序的类？这样做会不会导致项目的失败？我们该如何做才能让那些不是很优秀的程序员也能编写类？

4. 我们应该如何跟踪设计中的迭代，以确保我们能够作出正确的修正？